IT 1204
Section 3.0

Boolean Algebra and Digital Logic

Boolean Algebra

Logic Equations to Truth Tables

$$X = A.\overline{B} + \overline{A}.B + AB$$

A	B	X
0	0	
0	1	
1	0	
1	1	

Sum of Products

- The **OR** operation performed on the products of the **AND** operation
- Fill the corresponding cells with **1** for each product, the other cells with **0**

$$X = (A.B) + (\overline{A}.\overline{B}) + (\overline{A}.B)$$

$$A = 1, \overline{A} = 0$$

$$B = 1, \overline{B} = 0$$

A	B	X
0	0	1
0	1	1
1	0	0
1	1	1

Product of Sums

- The **AND** operation performed on the sums of the **OR** operation
- Fill the corresponding cells with **0** for each sum, the other cells with **1**

$$Y = (\overline{A} + \overline{B}).(\overline{A} + B)$$

$$A = 0, \overline{A} = 1$$

$$B = 0, \overline{B} = 1$$

A	B	X
0	0	1
0	1	1
1	0	0
1	1	0

Truth Tables to Logic Equations

A	B	X
0	0	1
0	1	1
1	0	0
1	1	0

- Sum of Products - consider 1s
  - Consider A=1,B=1

$$X = (\overline{A}.\overline{B}) + (\overline{A}.B)$$

- Product of sums – consider 0s
  - Consider A=0,B=0

$$X = (\overline{A} + B).(\overline{A} + \overline{B})$$

Your Turn: Exercise1

- Convert the following equation which is in the form of Product-of-sums into the form of Sum-of-products

$$f(ABCD) = (A + \overline{B} + C)(\overline{A} + B + \overline{C} + \overline{D})(\overline{A} + \overline{B} + D)(A + \overline{C})$$

Answer: Exercise1

$$f(ABCD) = (A + \overline{B} + C)(\overline{A} + B + \overline{C} + \overline{D})(\overline{A} + \overline{B} + D)(A + \overline{C})$$

$$f(ABCD) = \overline{\overline{(A + \overline{B} + C)(\overline{A} + B + \overline{C} + \overline{D})(\overline{A} + \overline{B} + D)(A + \overline{C})}}$$

$$f(ABCD) = \overline{(A + \overline{B} + C)} + \overline{(\overline{A} + B + \overline{C} + \overline{D})} + \overline{(\overline{A} + \overline{B} + D)} + \overline{(A + \overline{C})}$$

$$f(ABCD) = \overline{(\overline{A}.\overline{\overline{B}}.\overline{C}) + (\overline{\overline{A}}.B.\overline{\overline{C}}.\overline{\overline{D}}) + (\overline{\overline{A}}.\overline{\overline{B}}.\overline{D}) + (\overline{A}.\overline{\overline{C}})}$$

$$f(ABCD) = \overline{(\overline{A}.B.\overline{C}) + (A.\overline{B}.C.D) + (A.B.\overline{D}) + (\overline{A}.C)}$$

Boolean Postulates

0 . 0 = 0

1 + 1 = 1

0 + 0 = 0

1 . 1 = 1

1 . 0 = 0 . 1 = 0

1 + 0 = 0 + 1 = 1

Laws of Boolean Algebra

- **Commutative Law**
  - $A + B = B + A$
  - $A\ B = B\ A$

- **Associate Law**
  - $(A + B) + C = A + (B + C)$
  - $(A\ B)\ C = A\ (B\ C)$

Laws of Boolean Algebra

- **Distributive Law**
  - $A\ (B + C) = A\ B + A\ C$
  - $A + (BC) = (A + B)\ (A + C)$

- **Identity Law**
  - $A + A = A$
  - $A\ .\ A = A$

Laws of Boolean Algebra

- **Redundancy Law**
  - $A + AB = A$
  - $A\,(A + B) = A$

- **Demorgan's Theorem**
  - $\overline{(A + B)} = \overline{A}.\overline{B}$
  - $\overline{(A.B)} = \overline{A} + \overline{B}$

Laws of Boolean Algebra

- $A.B + A.\overline{B} = A$

  $(A + B)(A + \overline{B}) = A$

- $A + 0 = A$

  $A.0 = 0$

Laws of Boolean Algebra

- $1 + A = 1$

  $1 . A = A$

- $A + \overline{A} = 1$

  $A . \overline{A} = 0$

Laws of Boolean Algebra

- $A + \overline{A}.B = A + B$

$A(\overline{A} + B) = AB$

Implementation of Boolean Functions

- A Boolean function can be realised in either **SOP** or **POS** form
- At this point, it would seem that the choice would depend on whether the truth table contains more **1**s and **0**s for the output function
- The **SOP** has one term for each **1**, and the **POS** has one term for each **0**

Implementation of Boolean Functions

- However, there are other considerations:
  - It is generally possible to derive a simpler Boolean expression from truth table than either **SOP** or **POS**
  - It may be preferable to implement the function with a single gate type (NAND or NOR)

Implementation of Boolean Functions

- The significance of this is that, with a simpler Boolean expression, fewer gates will be needed to implement the function
- Methods that can be used to achieve simplification are:
  - Algebraic Simplification
  - Karnaugh Maps

Your Turn: Algebraic Simplification

- Simplify the following equation using Boolean algebra laws

$$f(ABC) = (A + \overline{B} + \overline{C})(A + \overline{B}C)$$

Answer: Algebraic Simplification

$$f(ABC) = (A + \overline{B} + \overline{C})(A + \overline{B}C)$$

$$f(ABC) = AA + A\overline{B}C + A\overline{B} + \overline{B}\,\overline{B}C + A\overline{C} + \overline{B}C\overline{C}$$

$$f(ABC) = A(1 + \overline{B}C + \overline{B} + \overline{C}) + \overline{B}C + \overline{B}C\overline{C}$$

$$f(ABC) = A + \overline{B}C$$

Karnaugh Maps

- For purposes of simplification, the Karnaugh map is a convenient way of representing a Boolean function of a small number (up to 4 to 6) of variables

- The map is an array of 2^n squares, representing the possible combinations of values of **n** binary variables

Karnaugh Maps

- The map can be used to represent any Boolean function in the following way:
  - Each square corresponds to a unique product in the **sum-of-products** form.
  - With a **1** value corresponding to the variable and a **0** value corresponding to the **NOT** of that variable

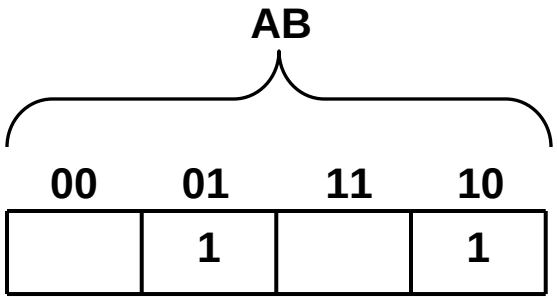
Karnaugh Maps: 2 Values

B \ A	0	1
0	0	1
1	1	1

$$X = A.\overline{B} + \overline{A}.B + AB$$

Karnaugh Maps: 2 Values

- The $A\bar{B}$ corresponds to the fourth square in the Figure
- For each such production in the function, 1 is placed in the corresponding square



$$F = A\bar{B} + \bar{A}B$$

Karnaugh Maps: 3 Values

C \ AB	00	01	11	10
0	1	1	0	0
1	0	0	1	1

$$X = A.\overline{B}.C + \overline{A}.B.\overline{C} + A.B.C + \overline{A}.\overline{B}.\overline{C}$$

Karnaugh Maps: 4 Values

CD \ AB	00	01	11	10
00	1	0	1	1
01	0	1	1	0
11	0	1	1	0
10	0	1	0	1

$$X = A.\overline{B}.\overline{C}.\overline{D} + A.\overline{B}.C.\overline{D} + A.B.\overline{C}.\overline{D} + \overline{A}.\overline{B}.\overline{C}.\overline{D} + \overline{A}.B.\overline{C}.D + A.B.\overline{C}.D + \overline{A}.B.C.D + A.B.C.D$$

Karnaugh Maps: Exercise 1

- Simplify the following Karnaugh Map using Boolean equations (Write your answers in both **SOP** and **POS**)

C \ AB	00	01	11	10
0	0	1	0	0
1	1	1	0	1

Karnaugh Maps: Answer

$$(\overline{A}.B.\overline{C}) + (\overline{A}.\overline{B}.C) + (\overline{A}.B.C) + (A.\overline{B}.C) \quad \Leftrightarrow \quad \overline{A}B + \overline{B}C$$

$$(A+B+C).(\overline{A}+\overline{B}+C).(\overline{A}+B+C).(\overline{A}+\overline{B}+\overline{C}) \quad \Leftrightarrow \quad (B+C).(\overline{A}+\overline{B})$$

C \ AB	00	01	11	10
0	0	1	0	0
1	1	1	0	1

Karnaugh Maps: Exercise 2

- Simplify the following Karnaugh Map using Boolean equations (Write your answers in both **SOP** and **POS**)

CD \ AB	00	01	11	10
00	1	0	0	1
01	0	1	1	0
11	0	0	1	0
10	1	0	0	1

Karnaugh Maps: Answer

$$(\overline{A}\overline{B}\overline{C}\overline{D}) + (\overline{A}\overline{B}\overline{C}D) + (\overline{A}\overline{B}C\overline{D}) + (\overline{A}BC\overline{D}) + (ABCD) + (A\overline{B}\overline{C}D) + (A\overline{B}C\overline{D})$$

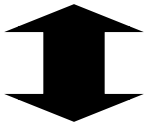
$$\Updownarrow$$

$$\overline{B}\overline{D} + B\overline{C}D + ABD$$

CD \ AB	00	01	11	10
00	1	0	0	1
01	0	1	1	0
11	0	0	1	0
10	1	0	0	1

Karnaugh Maps: Answer

$$(A+\overline{B}+C+D).(\overline{A}+\overline{B}+C+D).(A+\overline{B}+\overline{C}+D).(\overline{A}+\overline{B}+\overline{C}+D).$$
$$(A+B+C+\overline{D}).(A+B+\overline{C}+\overline{D}).(\overline{A}+B+C+\overline{D}).(\overline{A}+B+\overline{C}+\vec{D}).$$
$$(A+\overline{B}+\overline{C}+\overline{D})$$



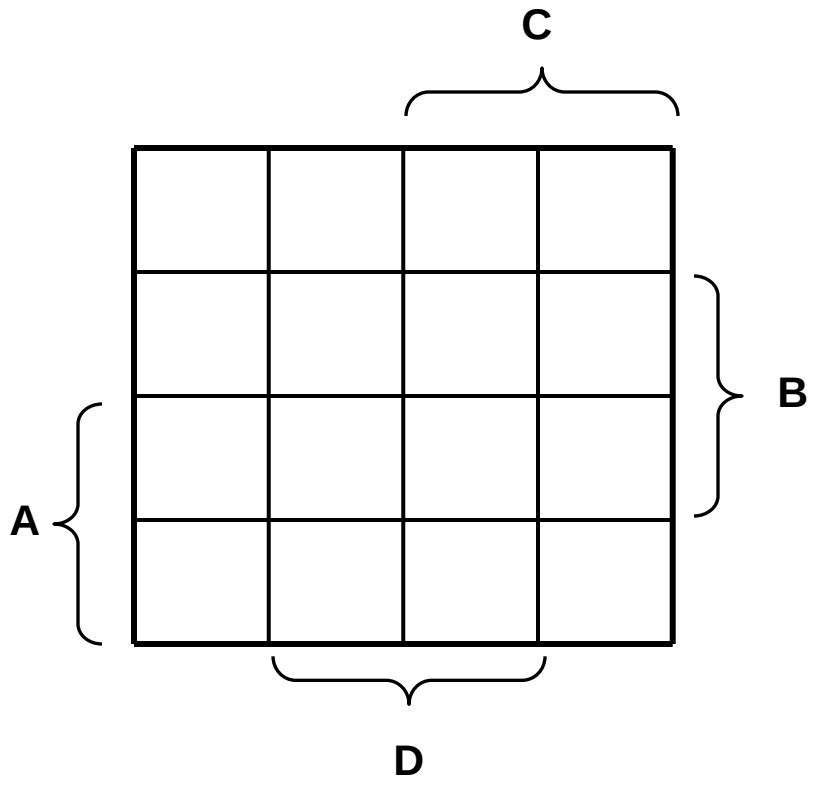
$$(\overline{B}+D).(B+\overline{D}).(A+\overline{C}+\overline{D})$$

CD \ AB	00	01	11	10
00	1	0	0	1
01	0	1	1	0
11	0	0	1	0
10	1	0	0	1

Simplified Labeling of Karnaugh Maps

- The labeling used in figure emphasizes the relationship between variables and the rows and columns of the map

  - The two rows embraced by the symbol **A** are those in which the variable **A** has the value **1**; the rows not embraced by the symbol **A** are those in which **A** is **0**



Simplified Labeling of Karnaugh Maps

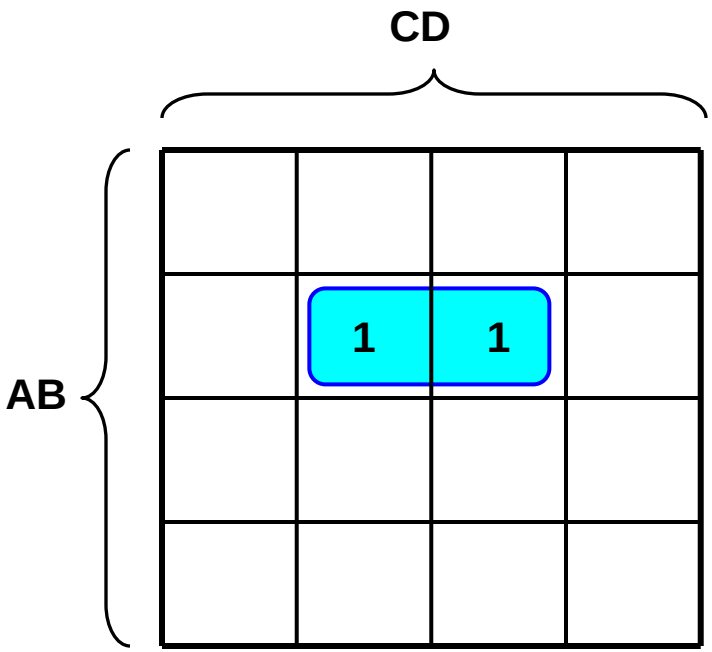
- Once the map of a function is created, we can often write a simple algebraic expression for it by noting the arrangement of the **1**s on the map
- The principle is as follows:
  - Any two squares that are adjacent differ in only one of the variables
  - If two adjacent squares both have an entry of **1**, then the corresponding product terms differ in **only one variable**
  - In such a case, the two terms can be merged by **eliminating that variable**

Simplified Labeling of Karnaugh Maps

- For example, in following FIGURE, the two adjacent squares correspond to the two terms $\bar{A}B\bar{C}D$ and $\bar{A}BCD$

- The function expressed is

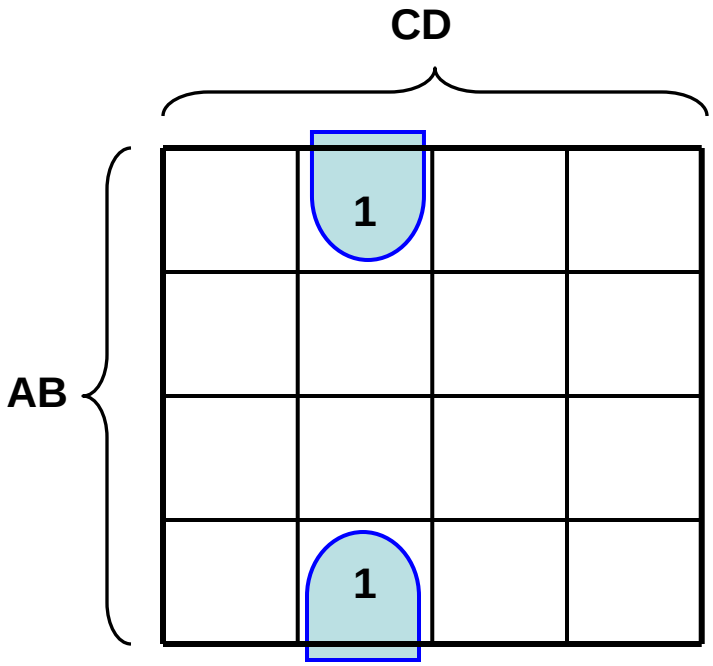
$\bar{A}B\bar{C}D + \bar{A}BCD = \bar{A}BD$

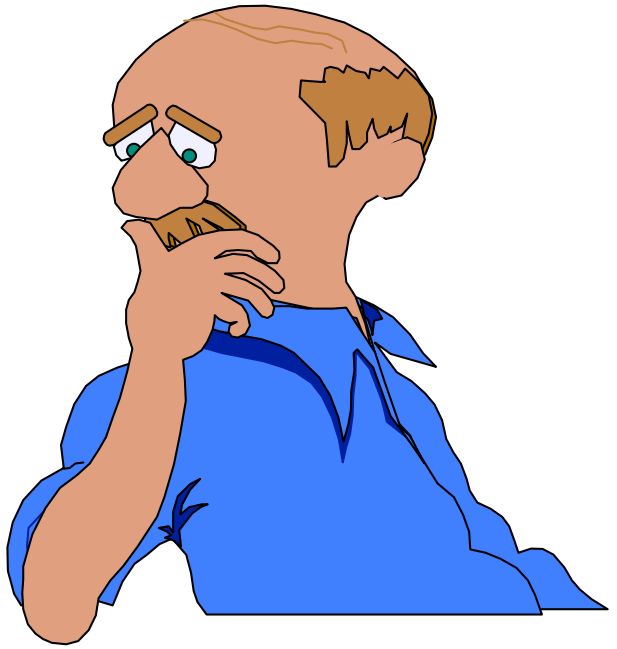


Simplified Labeling of Karnaugh Maps

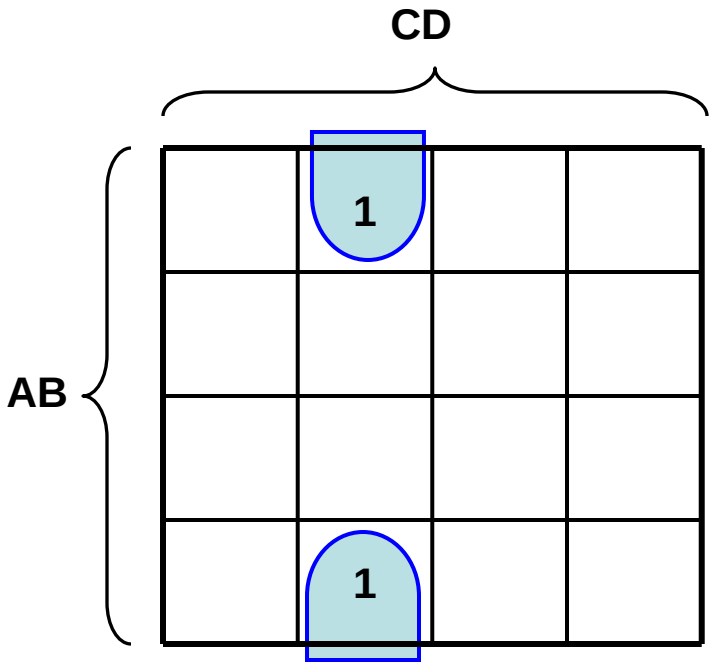
- This process can be extended in several ways:
  - First, the concept of adjacent can be extended to include wrapping around the edge of the map
  - Thus, the **top square of a column** is adjacent to the **bottom square**, and the **leftmost square of a row** is adjacent to the **rightmost square**
  - Second, we can group not just 2 squares but 2^n adjacent squares, that is, 4, 8, etc

Your turn: Karnaugh Maps



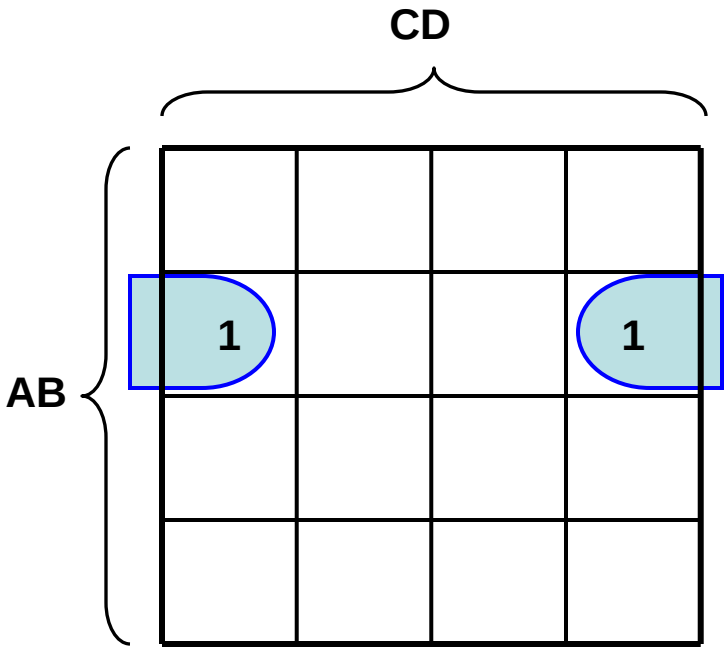


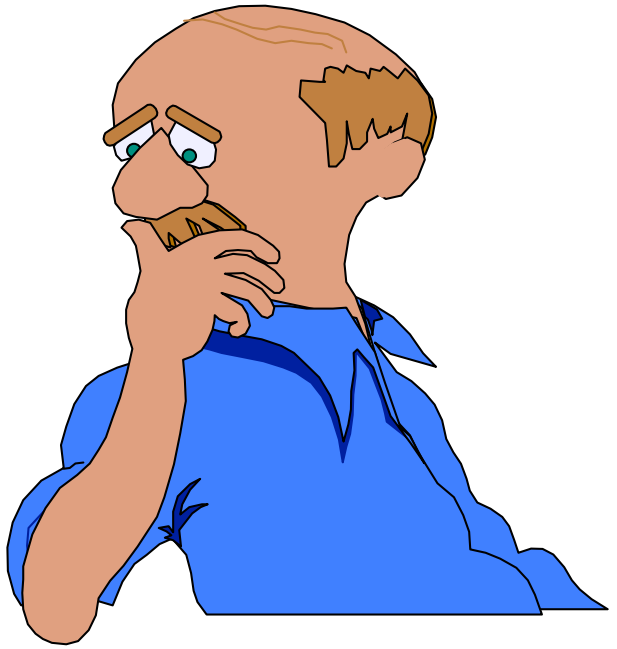
Answer: Karnaugh Maps



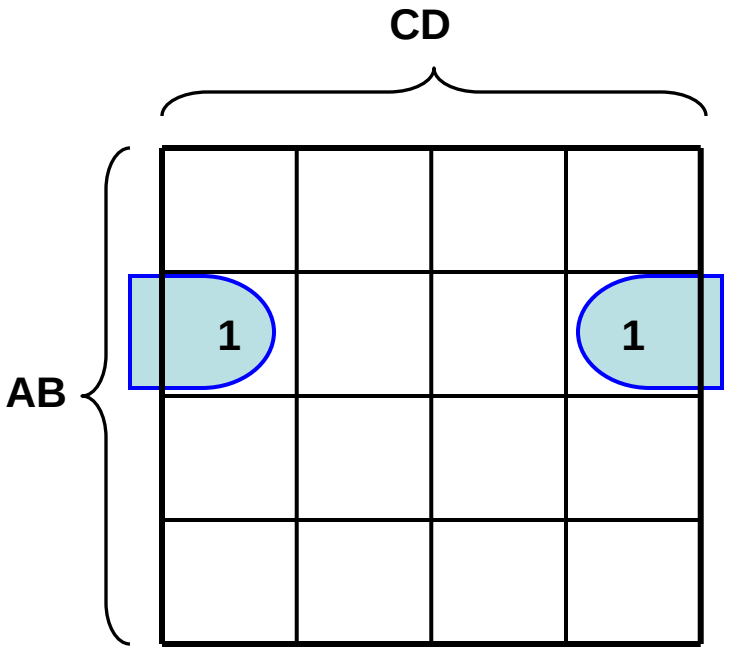
$\overline{B}\overline{C}D$

Your turn: Karnaugh Maps



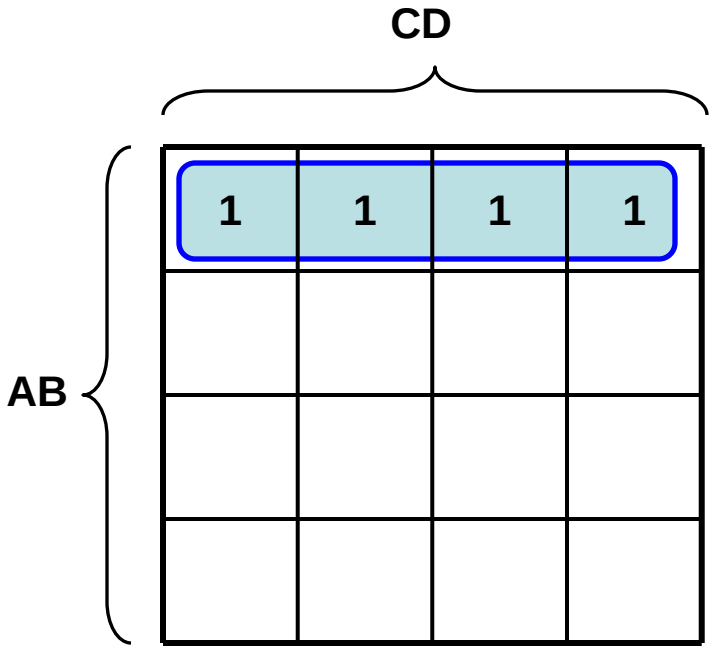


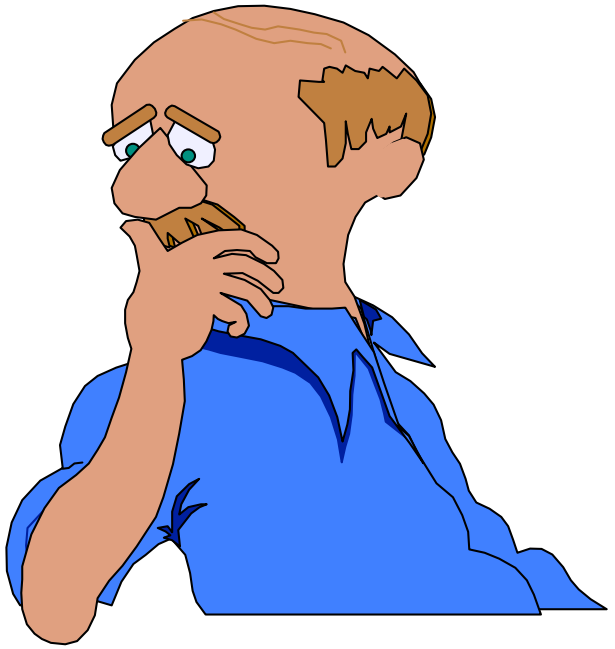
Answer: Karnaugh Maps



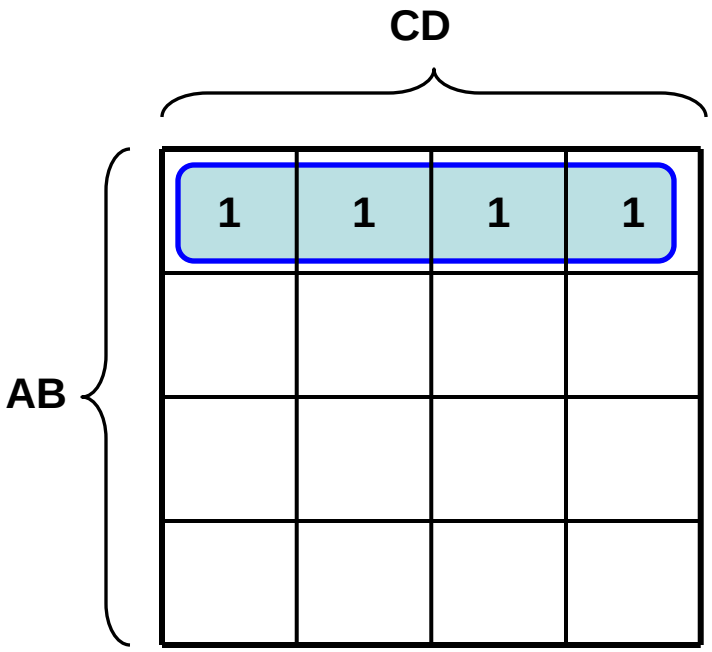
$\bar{A}B\bar{D}$

Your turn: Karnaugh Maps



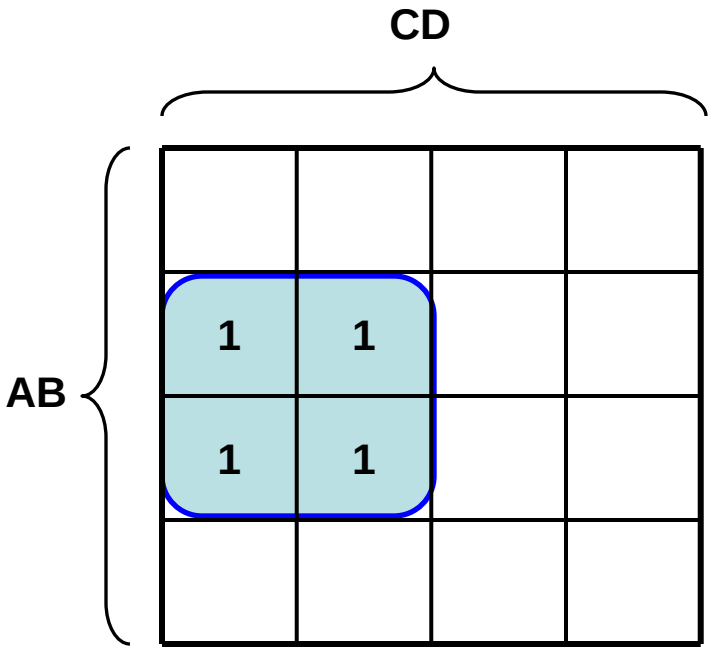


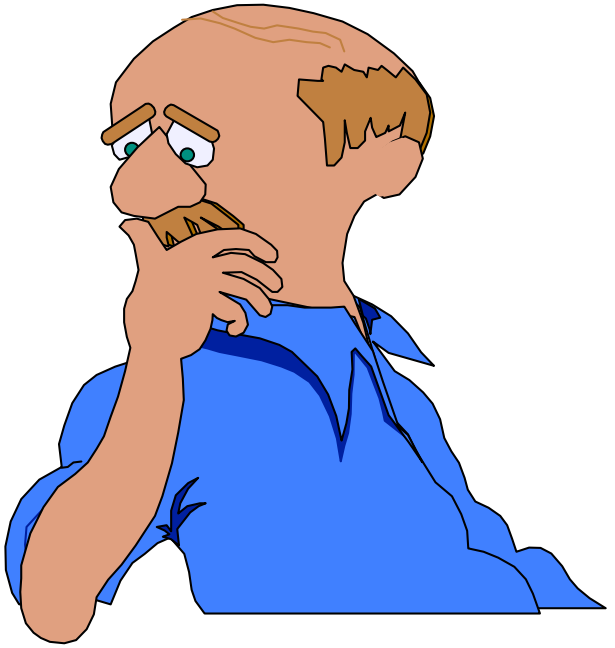
Answer: Karnaugh Maps



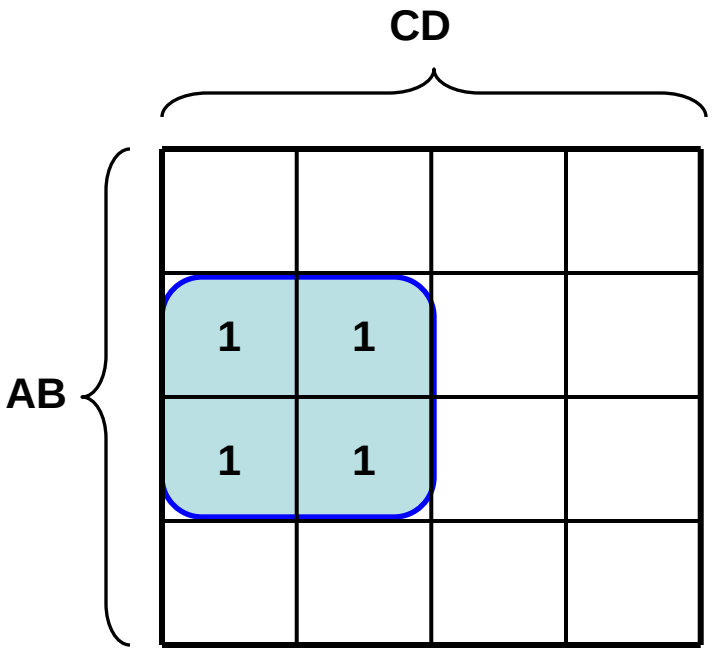
$\overline{A}\,\overline{B}$

Your turn: Karnaugh Maps



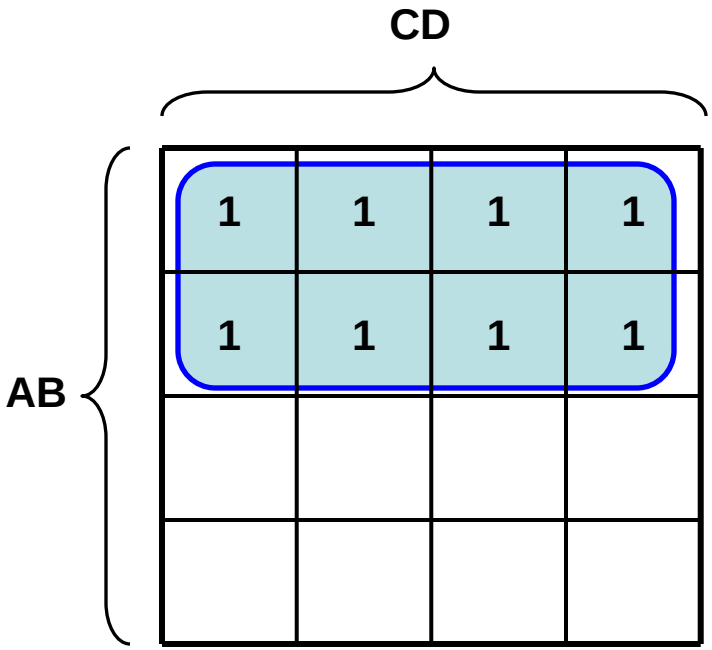


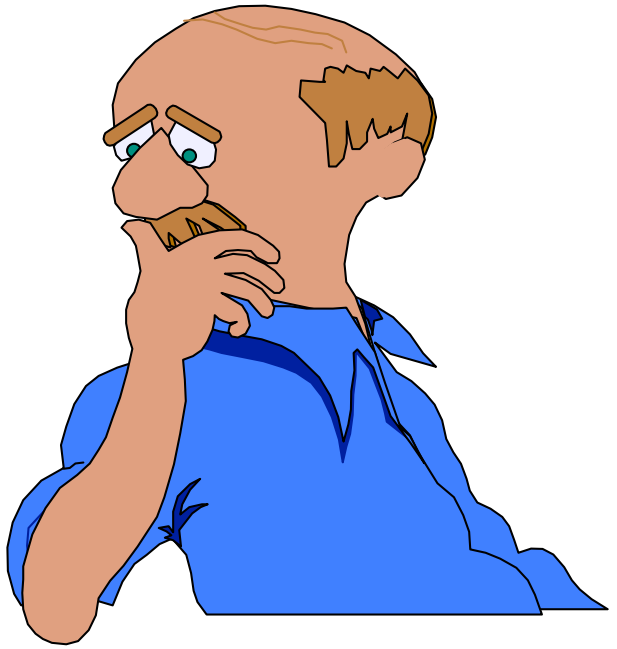
Answer: Karnaugh Maps



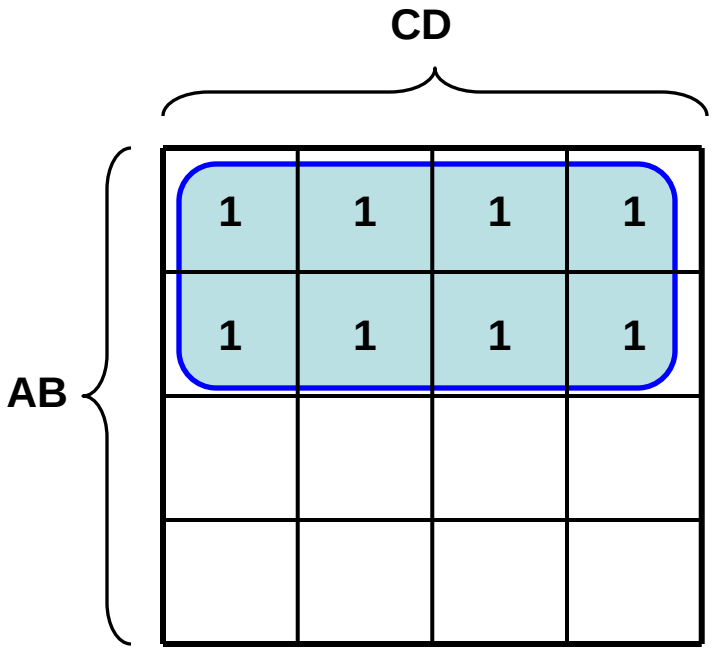
$B\overline{C}$

Your turn: Karnaugh Maps



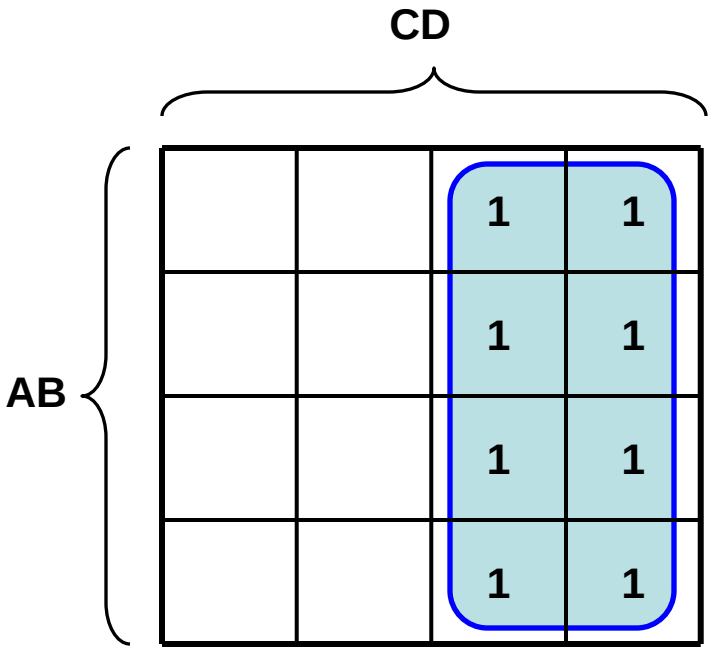


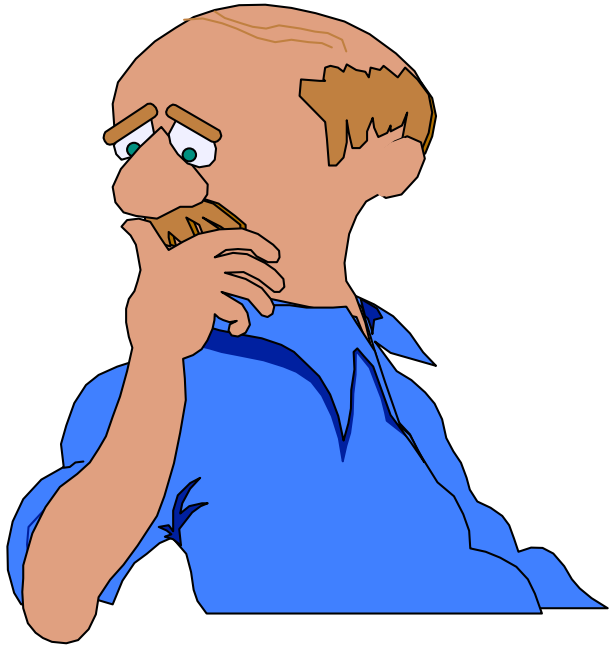
Answer: Karnaugh Maps



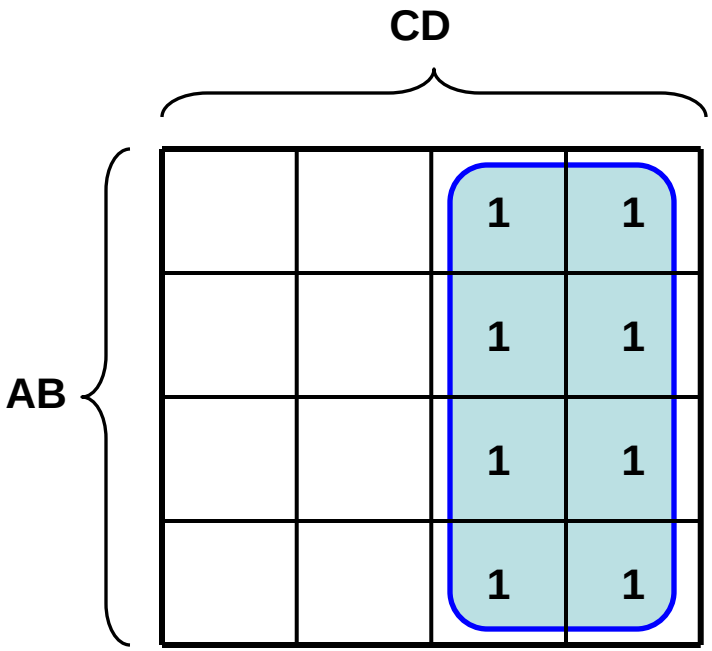
$\overline{A}$

Your turn: Karnaugh Maps





Answer: Karnaugh Maps

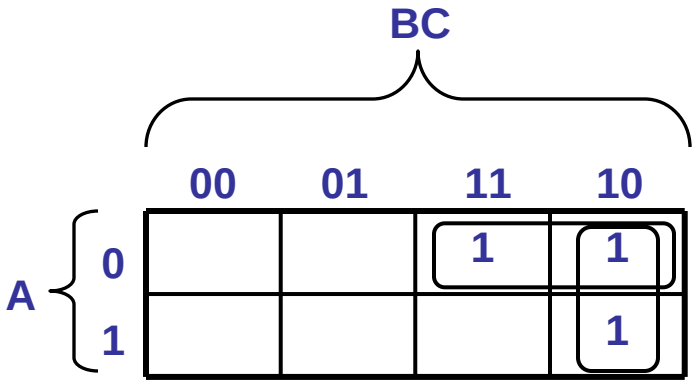


C

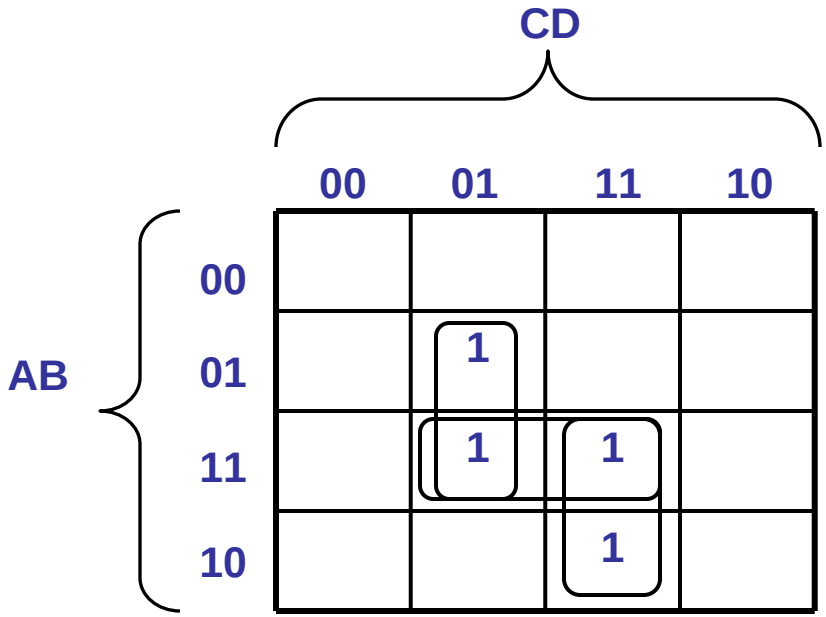
Simplified Labeling of Karnaugh Maps

- In attempting to simplify, first look for the largest grouping possible:
  - When you are circling groups, you are allowed to use the same **1** more than **once**
  - If any isolated **1**s remain after the groupings, then each of these is circled as a group of **1**s
  - Finally, before going from the map to a simplified Boolean expression, any group of **1**s that is **completely overlapped by other groups can be eliminated**

Karnaugh Maps: Overlapping Groups

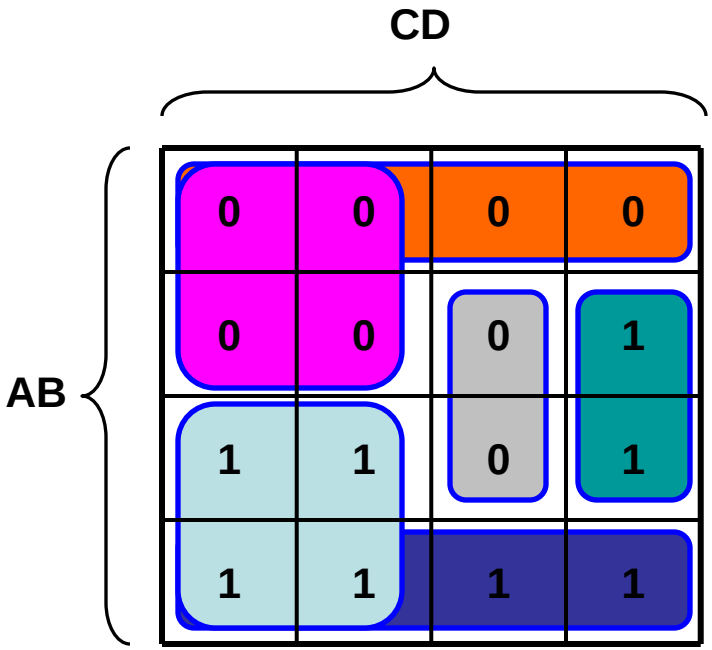


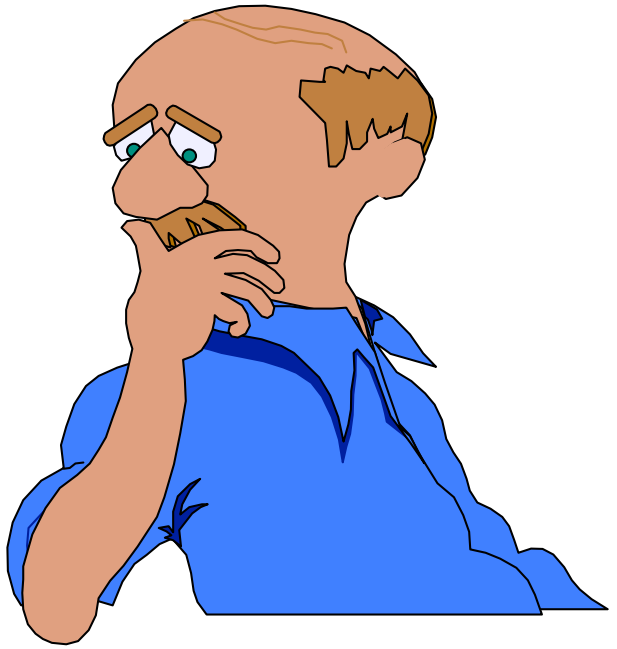
$F = \bar{A}B + B\bar{C}$



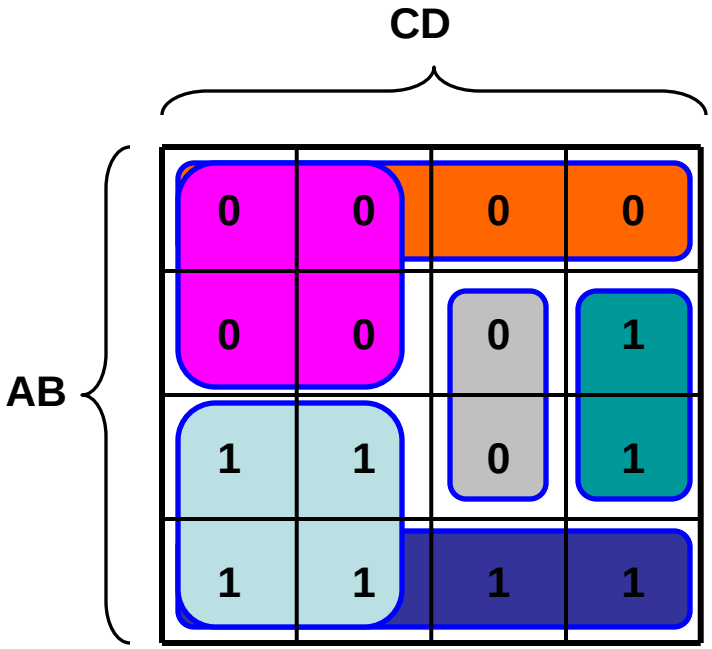
$F = B\bar{C}D + ACD$

Your turn: Karnaugh Maps





Answer: Karnaugh Maps



$$F = A\bar{C} + A\bar{B} + BC\bar{D}$$

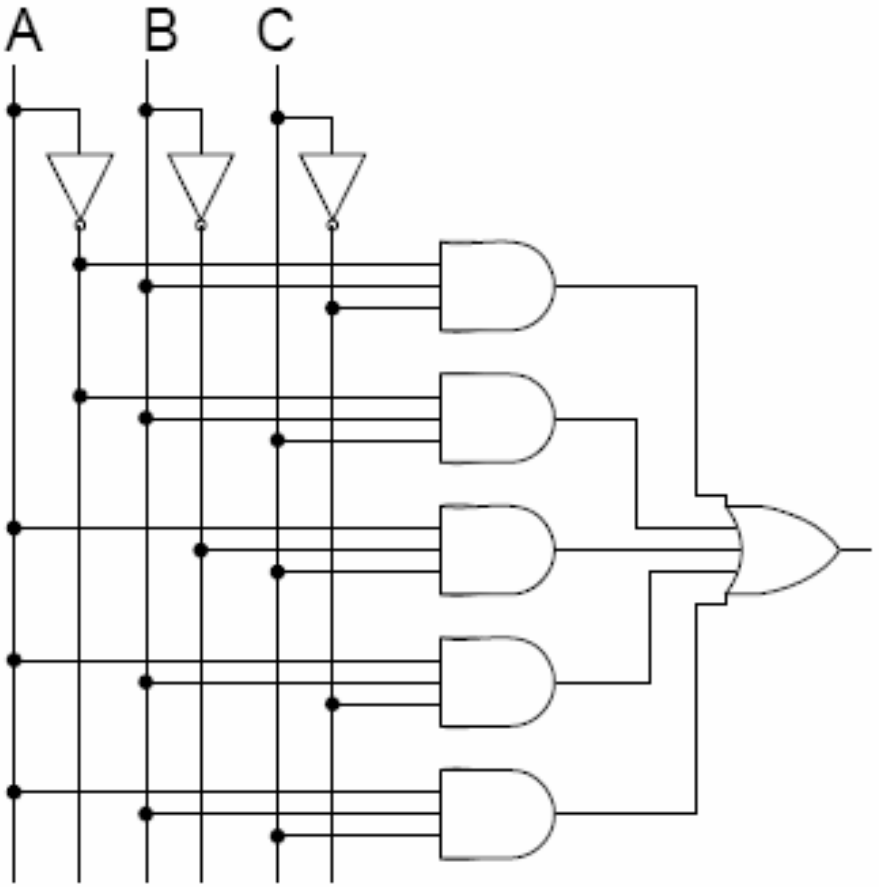
$$F = (A+C).(A+B).(\bar{B}+\bar{C}+\bar{D})$$

Drawing a Circuit

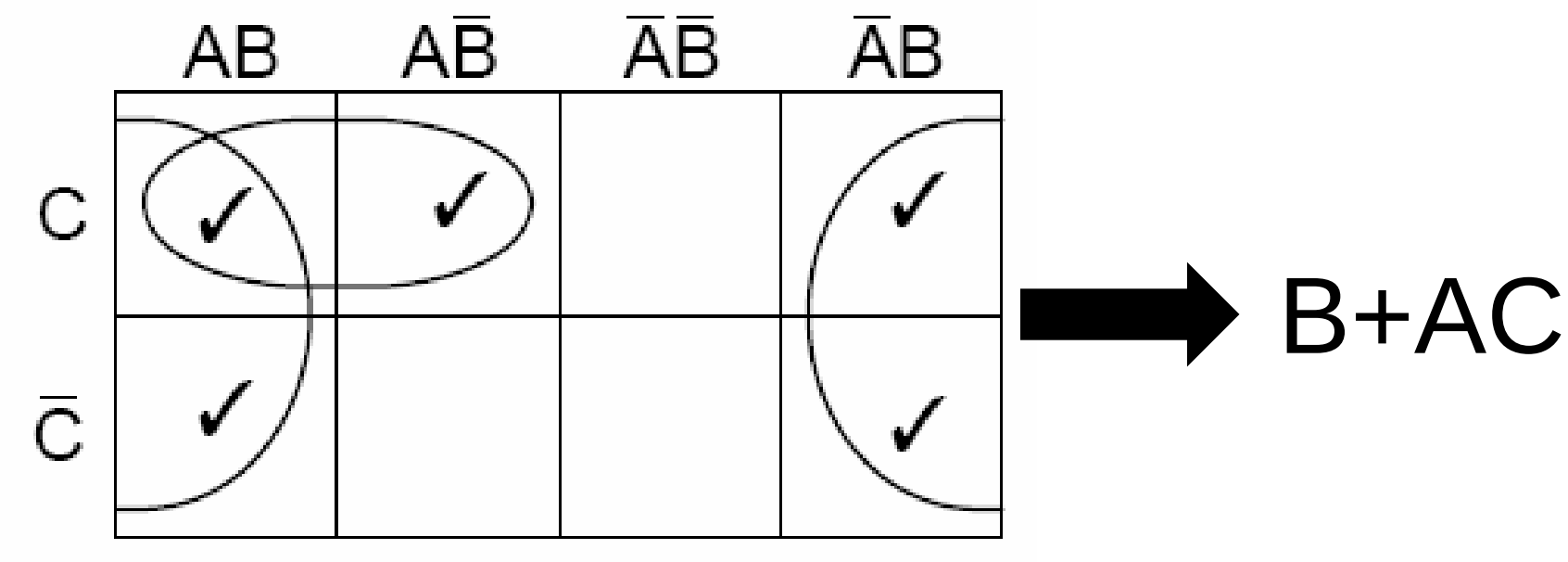
Sum-of-Products Expression

$$\overline{A}\,B\,\overline{C} + \overline{A}\,B\,C + A\,\overline{B}\,C + A\,B\,\overline{C} + A\,B\,C$$

Digital Logic Circuit

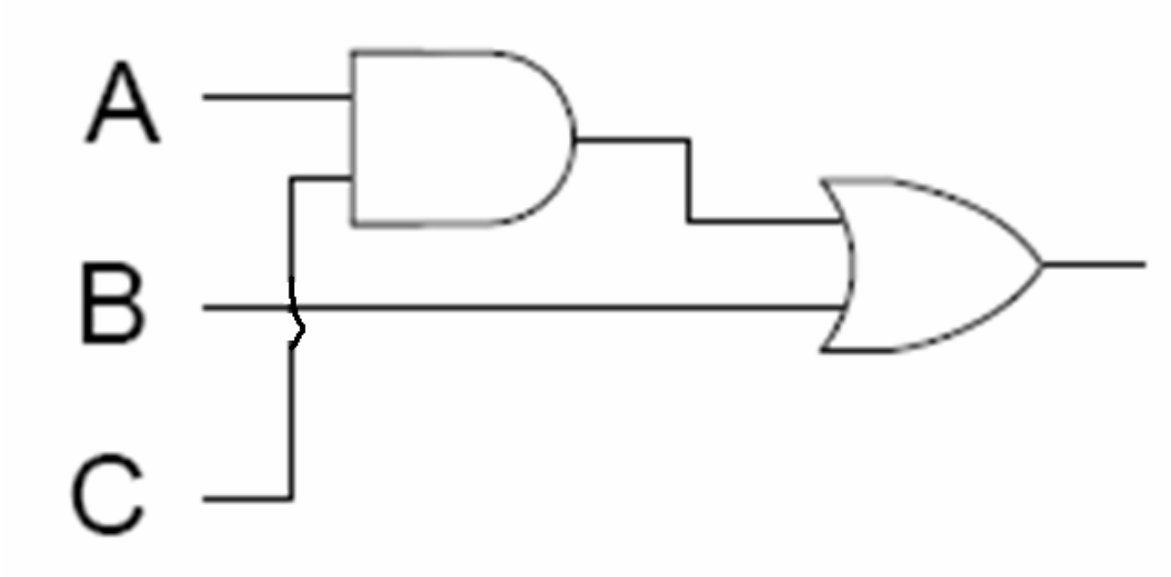


Drawing a Circuit



Drawing a Circuit

B+AC



Logic Operators

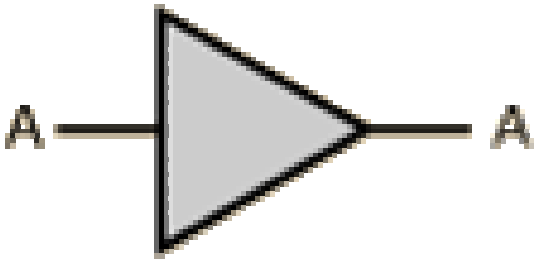
Logic Operations

- Basic logic operators and logic gates
- Boolean algebra
- Combinational Circuits
- Basic circuit design

Basic Logic Operators and Logic Gates

- AND
- OR
- NOT
- XOR (Exclusive OR)
- NOR
- NAND
- XNOR

Buffer



A	B
0	0
1	1

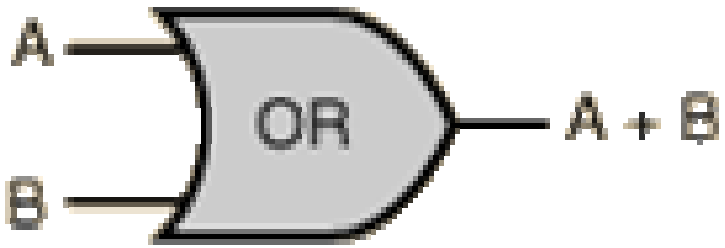
AND Operation

- . Operator
- ^ Operator
- **A . B = A ^ B**

A	B	A . B
0	0	0
0	1	0
1	0	0
1	1	1

OR Operation

- **+** Operator
- **v** Operator
- **A + B = A v B**



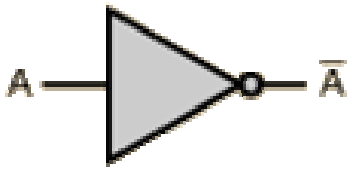
A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

NOT Operation

- ~ Operator
- ¬ Operator

$$\overline{A} = \neg A = \sim A = A'$$

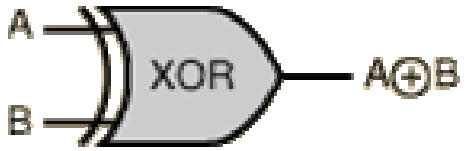
A	A’
0	1
1	0



XOR Operation

- $\oplus$ Operator

$$A \oplus B$$



A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

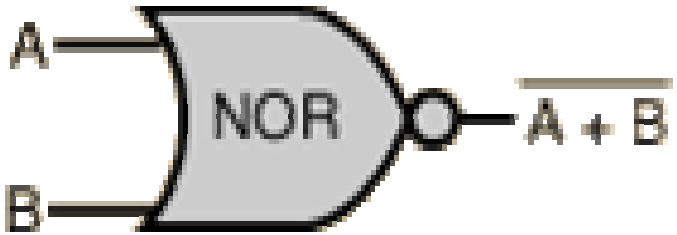
NAND Operation

$$(\overline{A.B}) = (A.B)'$$

A	B	$(\overline{A.B})$
0	0	1
0	1	1
1	0	1
1	1	0

NOR Operation

$$(\overline{A+B}) = (A+B)'$$



A	B	$(\overline{A+B})$
0	0	1
0	1	0
1	0	0
1	1	0

XNOR Operation

$$(\overline{A \oplus B})$$

A	B	$(\overline{A \oplus B})$
0	0	1
0	1	0
1	0	0
1	1	1

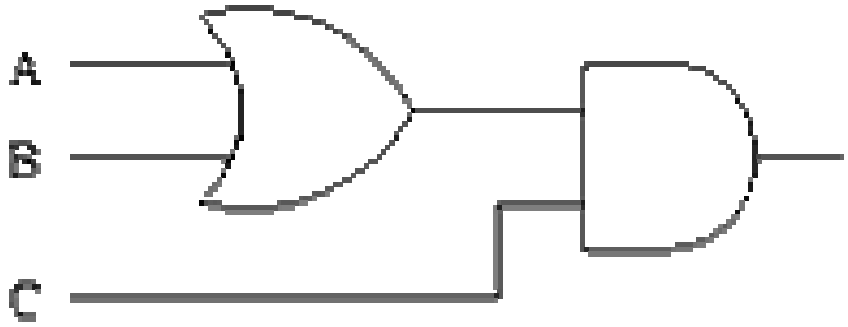
Drawing Logic Gates

- In addition to the basic gates, gates with 3,4, or more inputs can be used

  E.g. **x + y + z** can be implemented with a single **OR** gate with 3 inputs

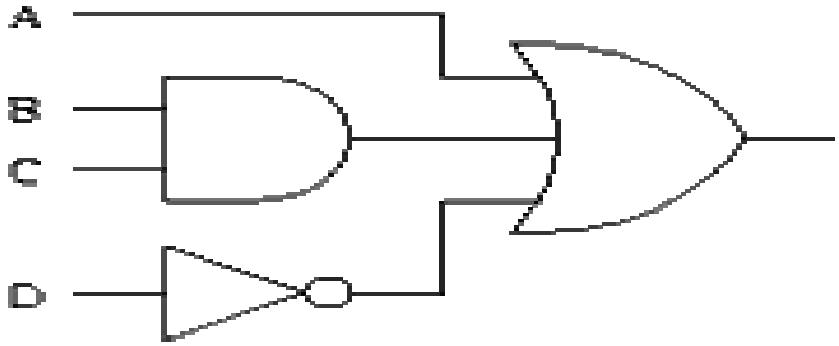
Drawing Logic Gates

$$X = (A + B)C$$



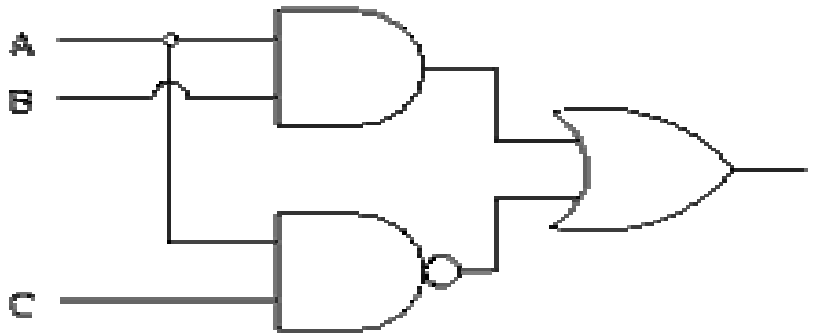
Drawing Logic Gates

$$X = A + (B.C) + \overline{D}$$



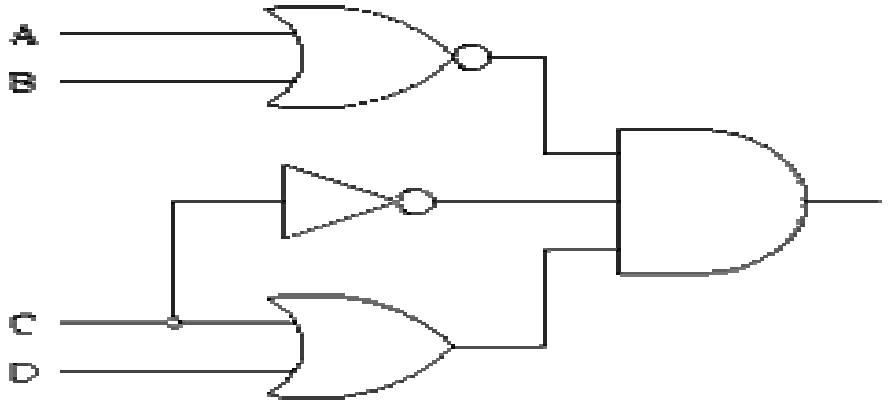
Drawing Logic Gates

$$X = (A.B) + (\overline{A.C})$$



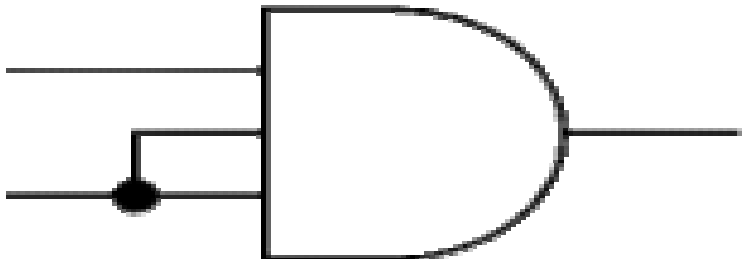
Drawing Logic Gates

$$X = \overline{(A + B)}.(C + D).\overline{C}$$



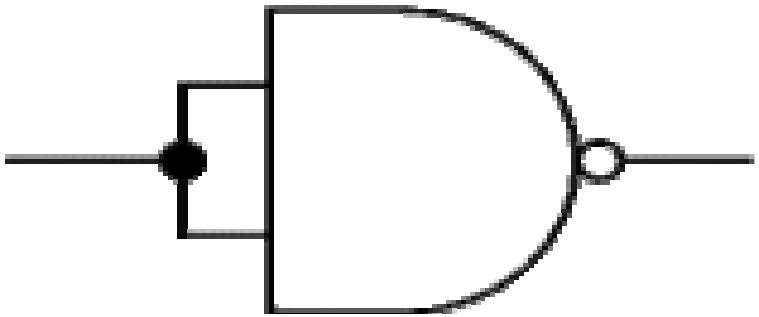
Reducing Logic Gates

- Reducing the number of inputs
  - The number of inputs to a gate can be reduced by connecting two (or more) inputs together
  - The diagram shows a 3-input **AND** gate operating as a 2-input **AND** gate



Reducing Logic Gates

- Reducing the number of inputs
  - Reducing a NOT gate from a NAND or NOR gate
  - The diagram shows this for a 2-input **NAND** gate



Logic Gates

- Typically, not all gate types are used in implementation
  - Design and fabrication are simpler if only one or two types of gates are used
  - Therefore, it is important to identify ***functionally complete*** sets of gates
  - This means that any **Boolean function** can be implemented using only the gates in the set

Logic Gates

- The following are functionally complete sets:
  - **AND, OR, NOT**
  - **AND, NOT**
  - **OR, NOT**
  - **NAND**
  - **NOR**

Logic Gates

- **AND**, **OR**, and **NOT** gates constitute a functionally complete set, since they represent the 3 operations of ***Boolean algebra***
- For the **AND** and **NOT** gates to form a functionally complete set, there must be a way to synthesize the **OR** operation from the **AND** and **NOT** operations

$$A + B = \overline{\overline{A} \cdot \overline{B}}$$

A OR B = NOT((NOT A) AND (NOT B))

Logic Gates

- Similarly, the **OR** and **NOT** operations are functionally complete because they can be synthesize the **AND** operation

$$A \cdot B = \overline{\overline{A} + \overline{B}}$$

A AND B = NOT((NOT A) OR (NOT B))

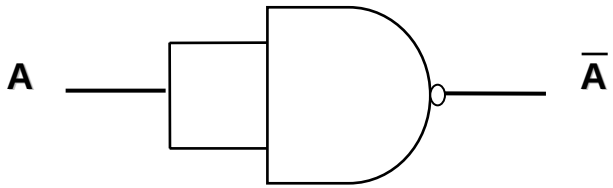
Logic Gates

- The **AND, OR** and **NOT** functions can be implemented solely with **NAND** gates, and the same thing for **NOR** gates.

- For this reason, digital circuits can be, and frequently are, implemented solely with **NAND** gates or solely with **NOR** gates

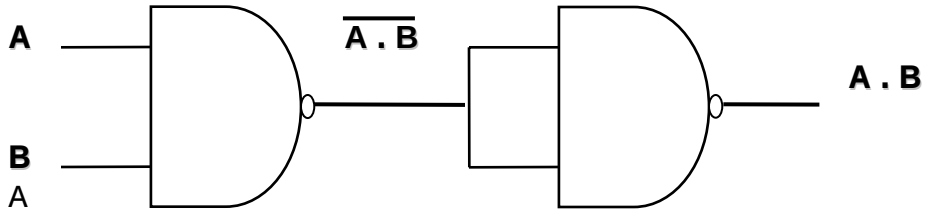
Logic Gates

- The diagram shows how the **NOT** function can be implemented solely with **NAND** gate



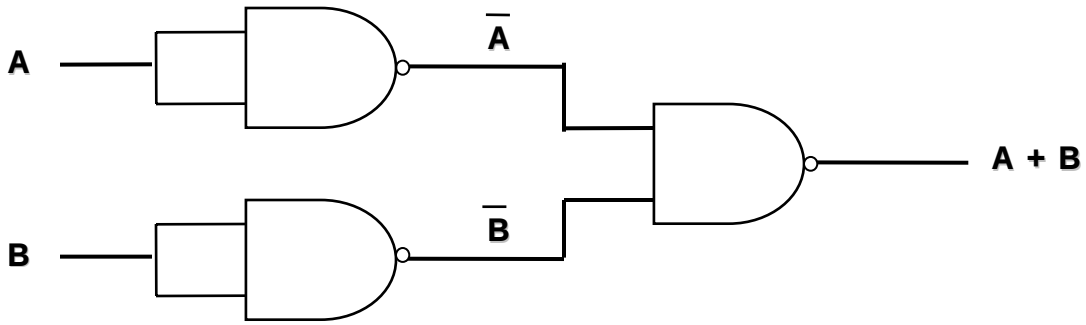
Logic Gates

- The diagram shows how the **AND** function can be implemented solely with **NAND** gate

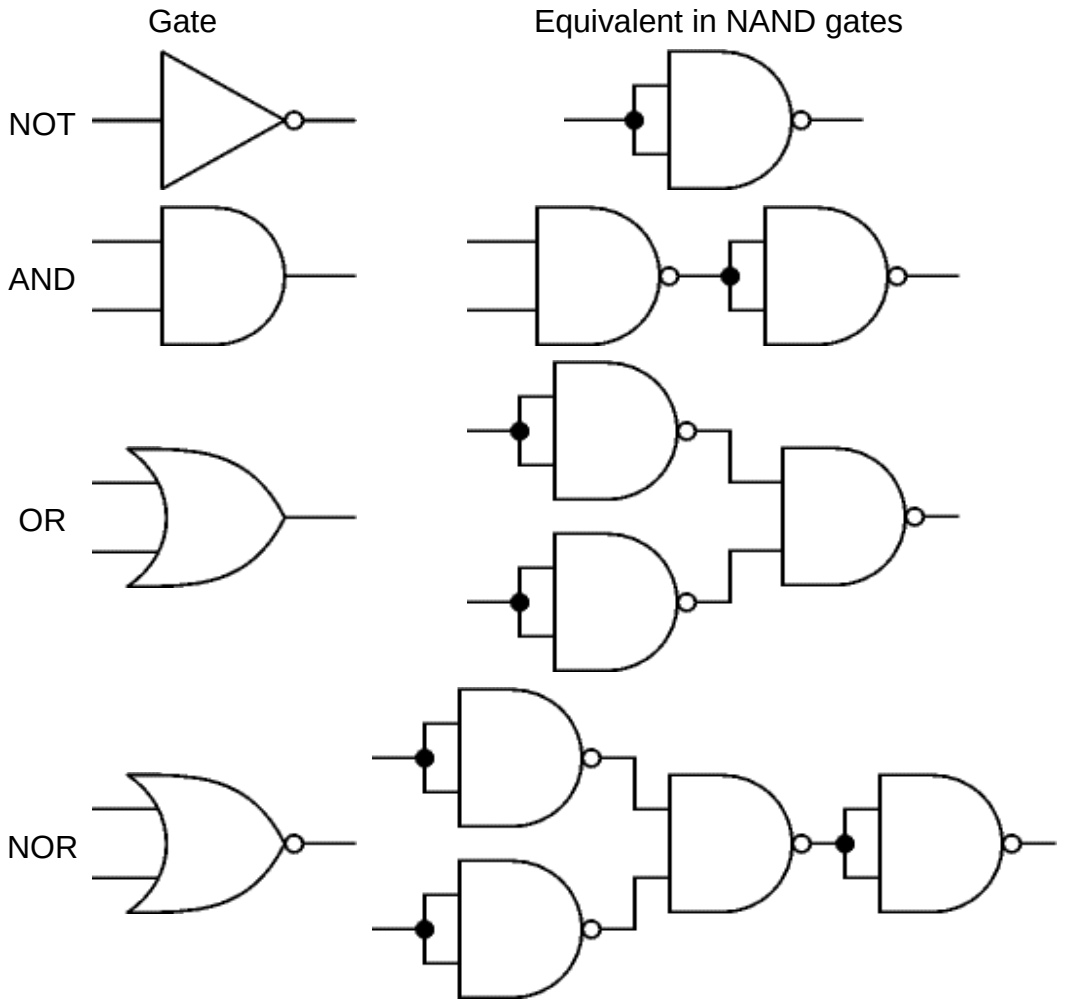


Logic Gates

- The diagram shows how the **OR** function can be implemented solely with **NAND** gate

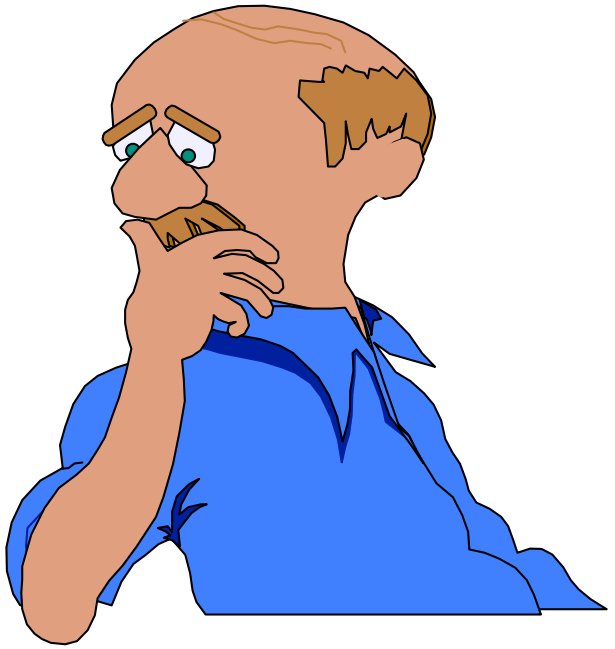


Logic Gates



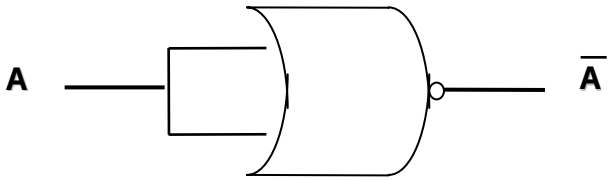
Your turn

- Draw a diagram that shows how the **NOT** function can be implemented solely with **NOR** gate



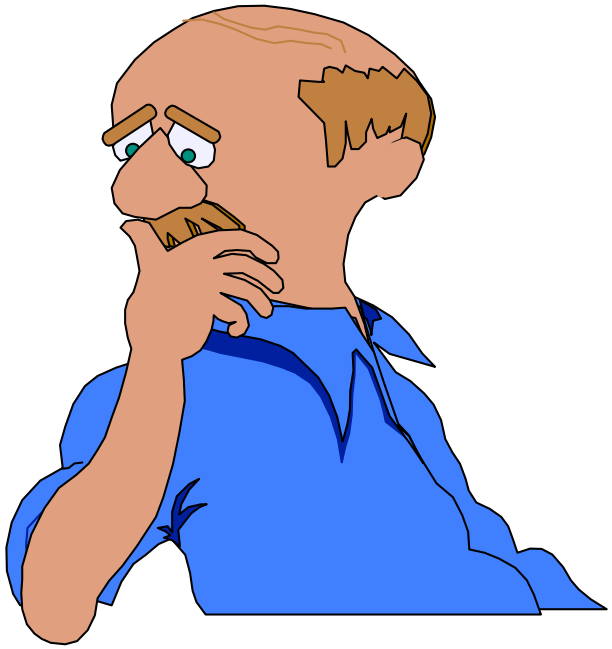
Logic Gates

- The diagram shows how the **NOT** function can be implemented solely with **NOR** gate



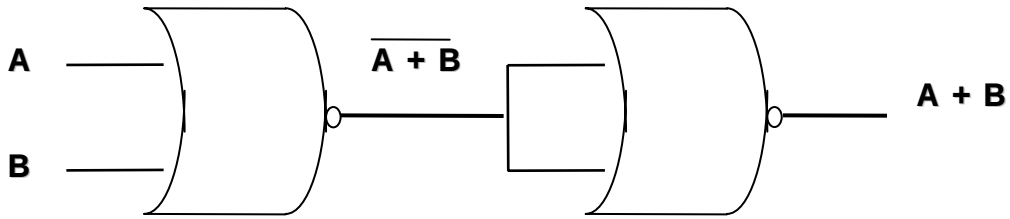
Your turn

- Draw a diagram that shows how the **OR** function can be implemented solely with **NOR** gate



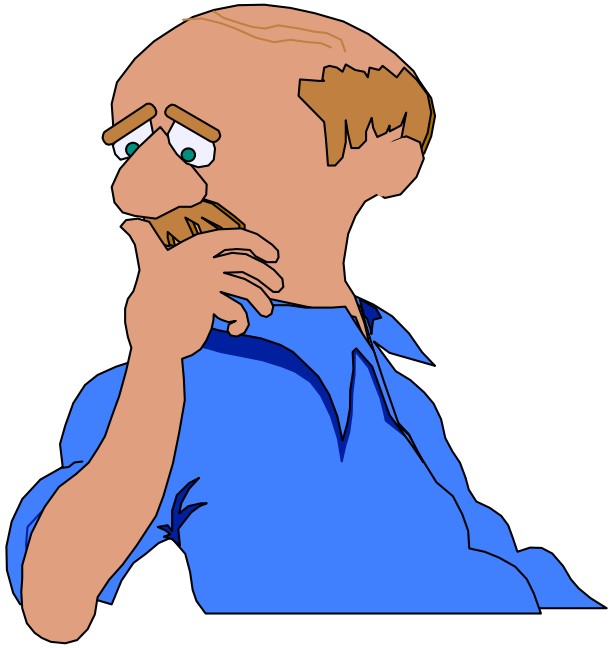
Logic Gates

- The diagram shows how the **OR** function can be implemented solely with **NOR** gate



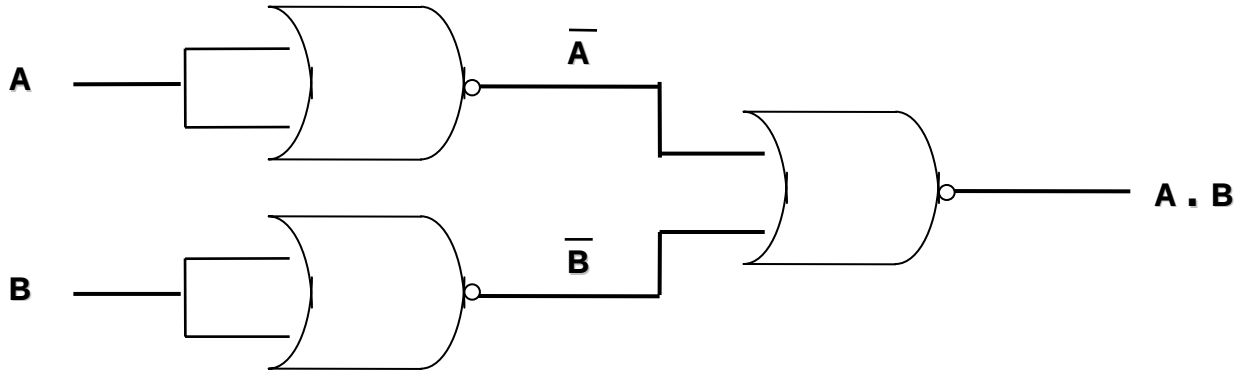
Your turn

- Draw a diagram that shows how the **AND** function can be implemented solely with **NOR** gate

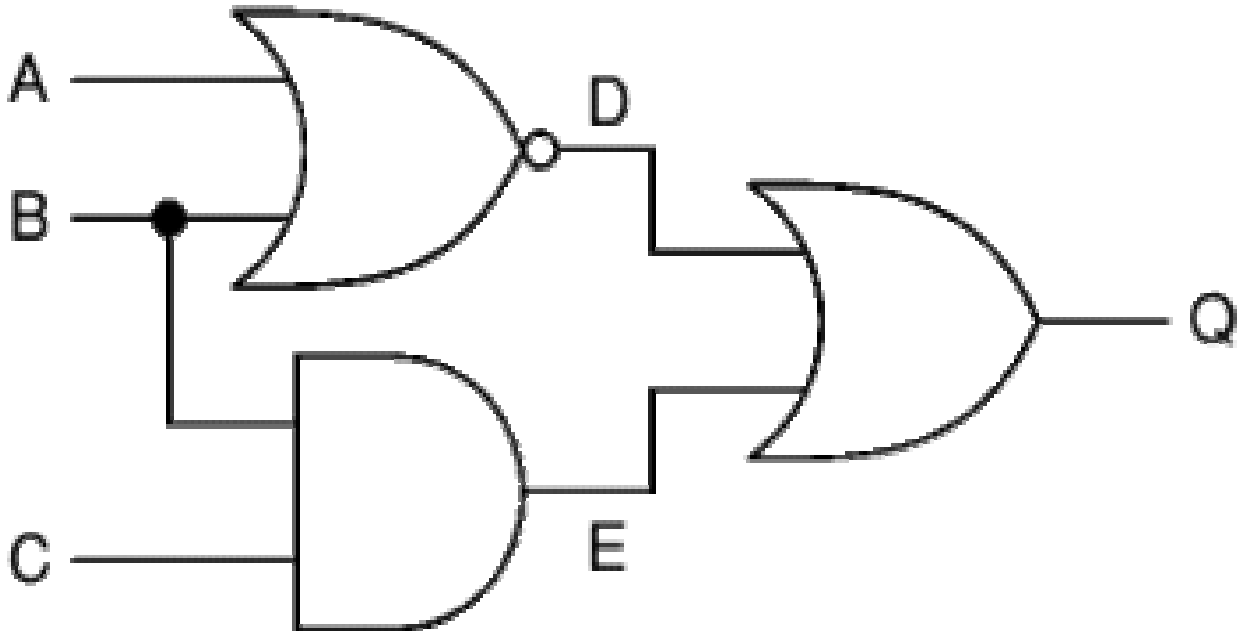


Logic Gates

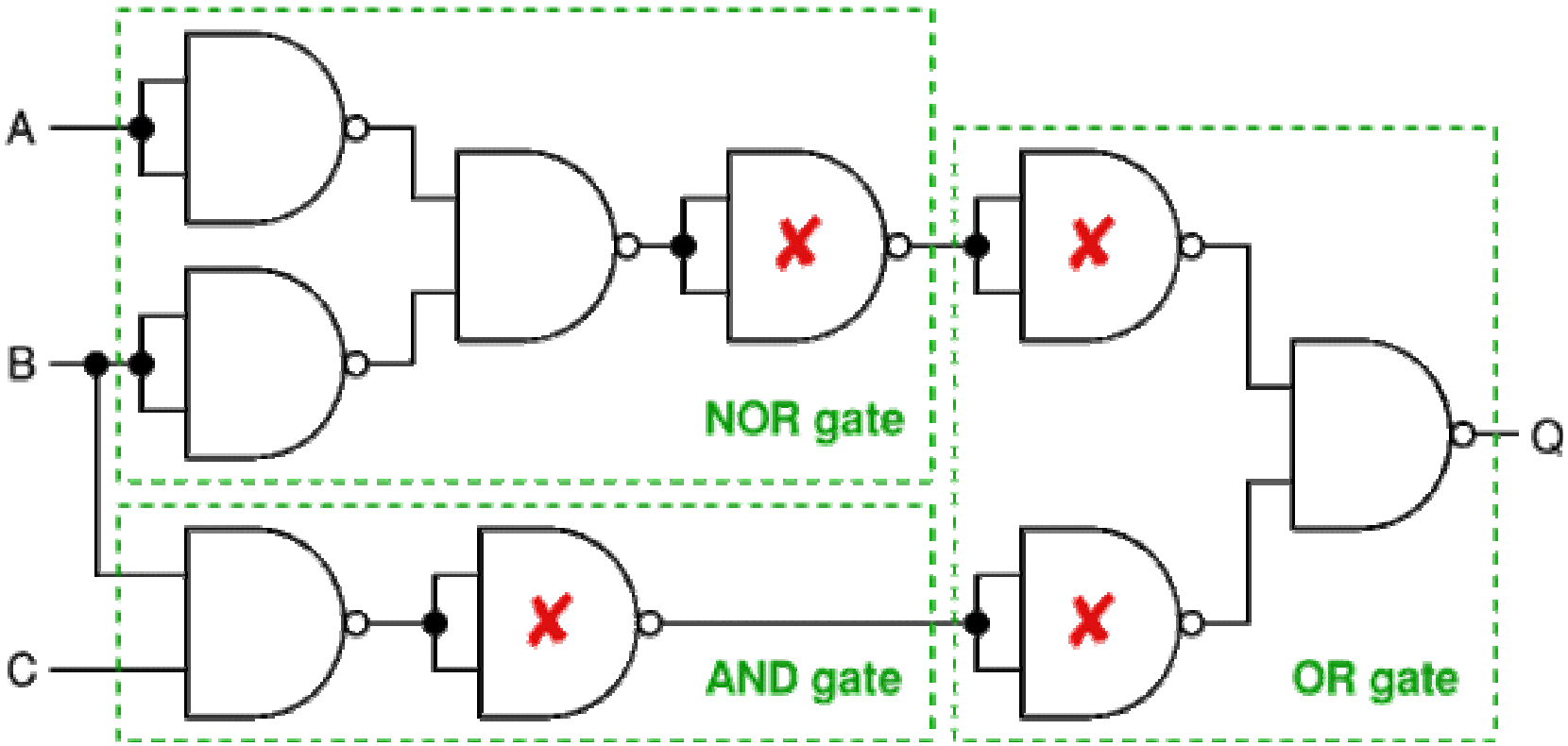
- The diagram shows how the **AND** function can be implemented solely with **NOR** gate



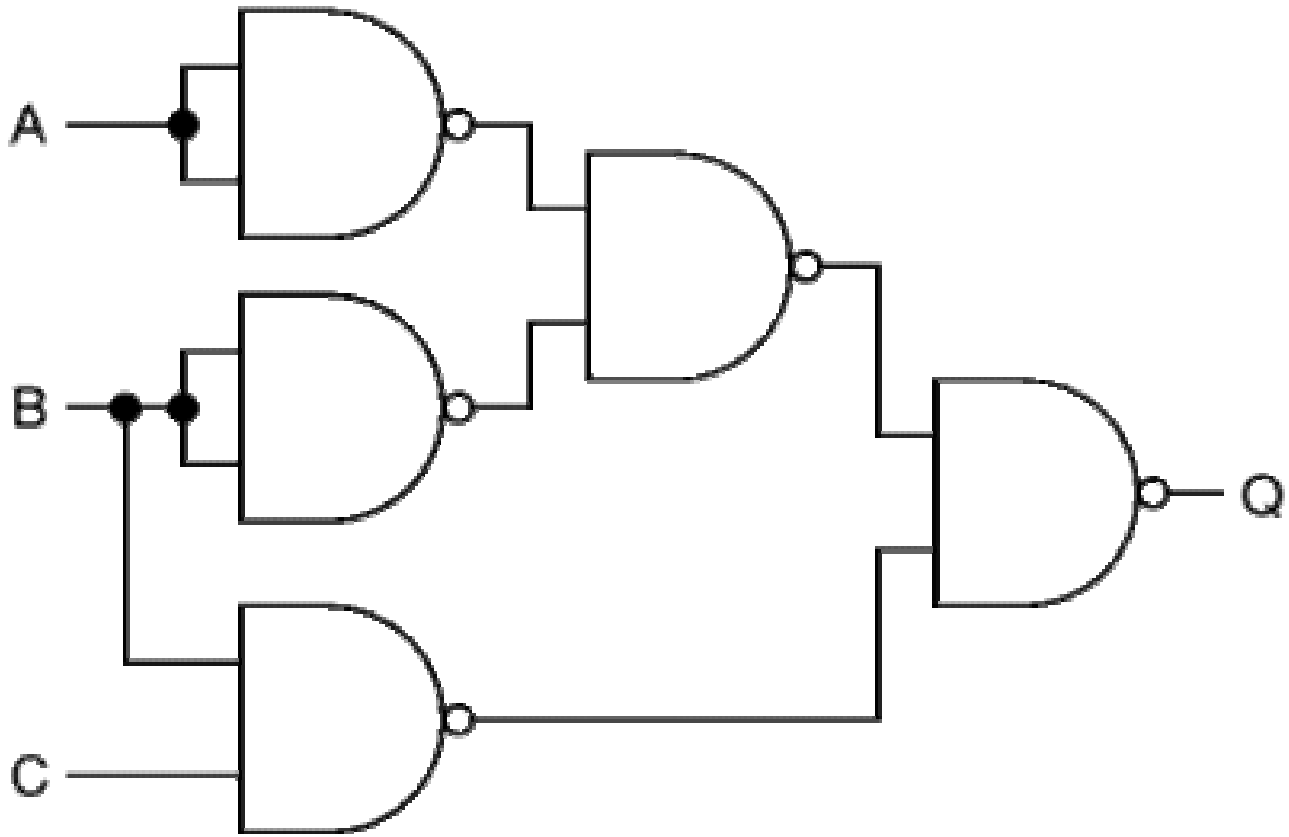
Substituting Gates in an Logic System



Substituting Gates in an Logic System



Substituting Gates in an Logic System



Combinational Circuits

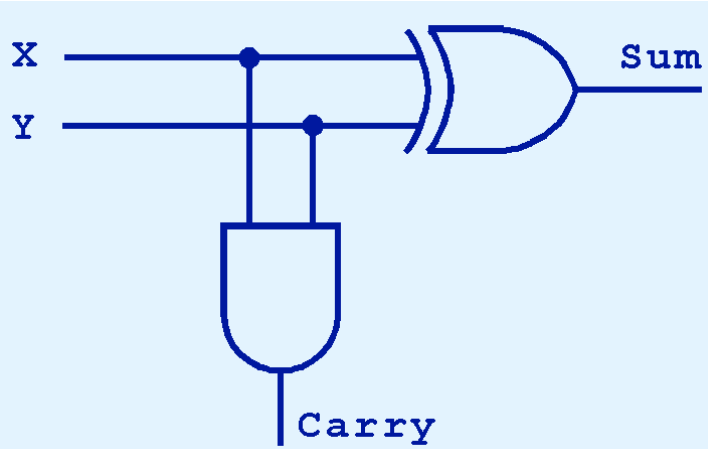
Combinational Logic

- Also called combinatorial logic
- A type of logic circuit whose output is a function of the present input only

Half Adder

- Finds the sum of two bits
- The sum can be found using the XOR operation and the carry using the AND operation

Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



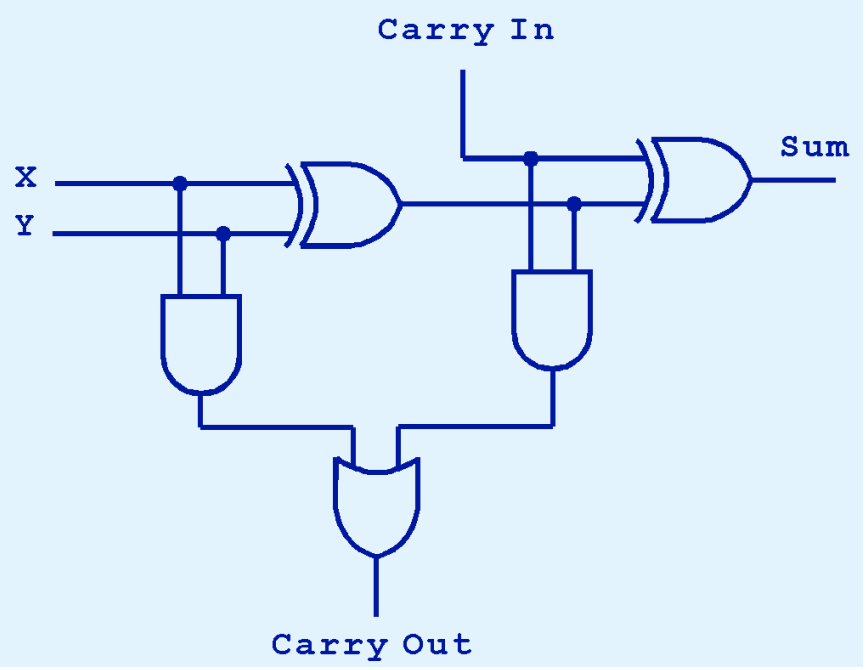
Full Adder

- We can change our half adder into to a full adder by including gates for processing the carry bit
- The truth table for a full adder is:

Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

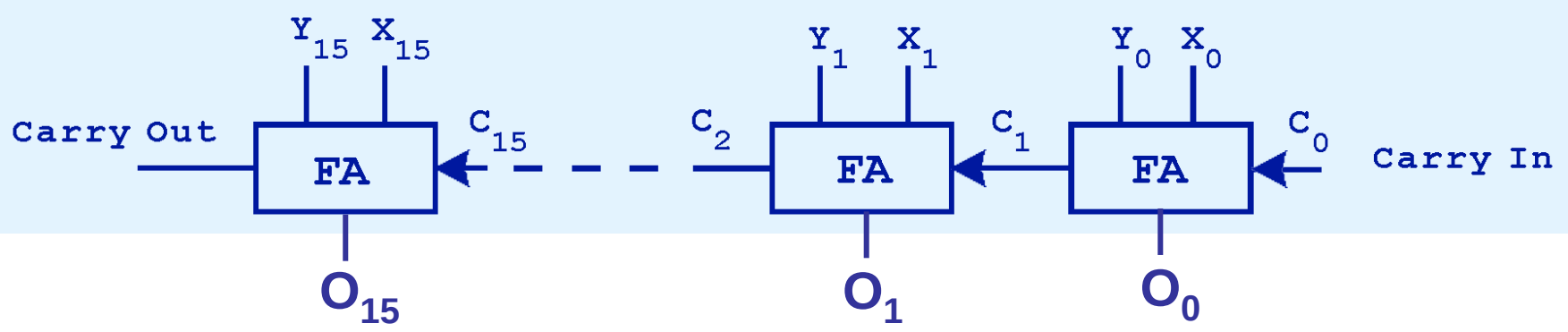
Converting a Half Adder into a Full Adder

Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

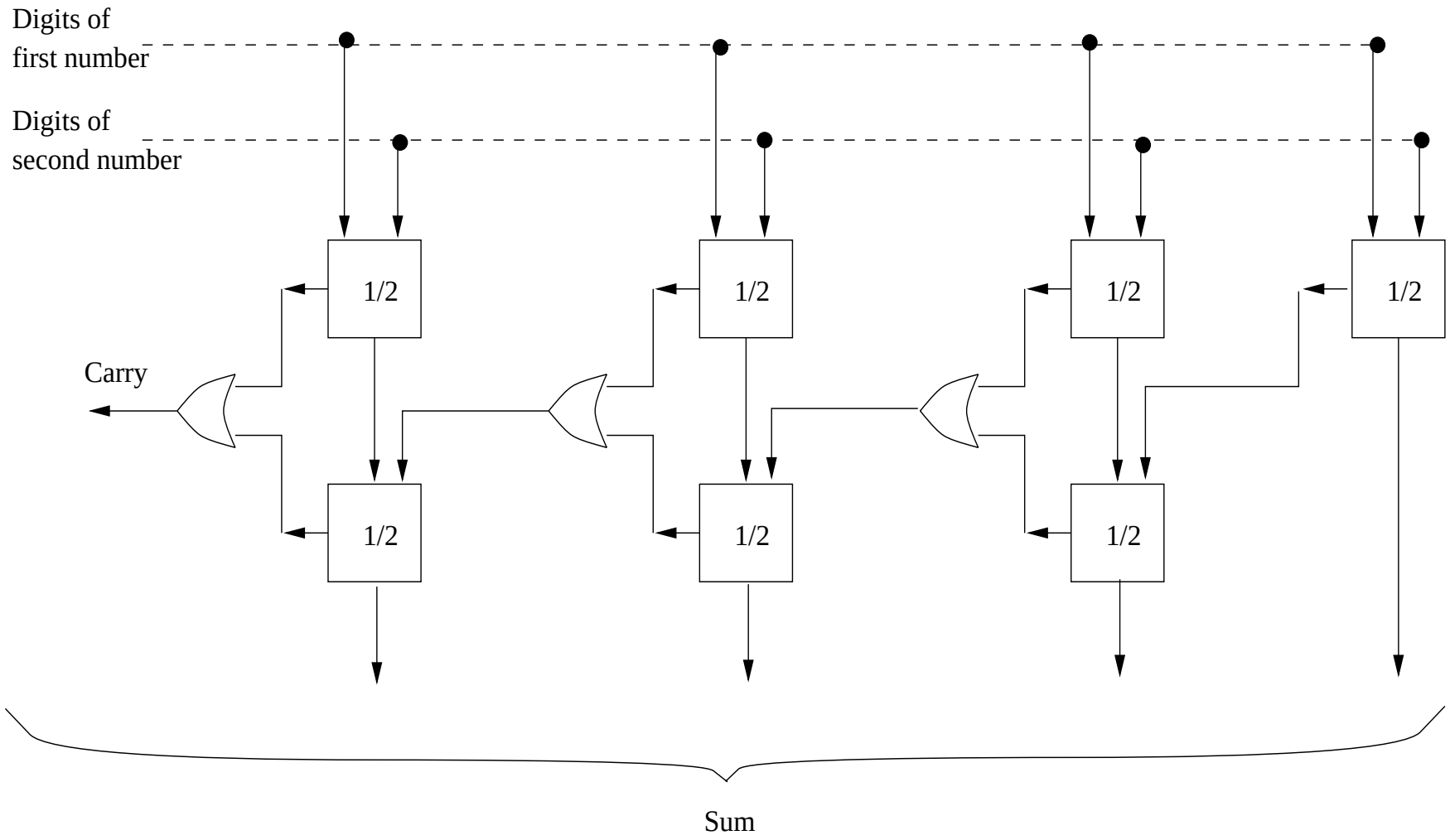


Ripple-carry Adder (I)

- Just as we combined half adders to make a full adder, full adders can be connected in series
  - The carry bit "ripples" from one adder to the next; hence, this configuration is called a ripple-carry adder

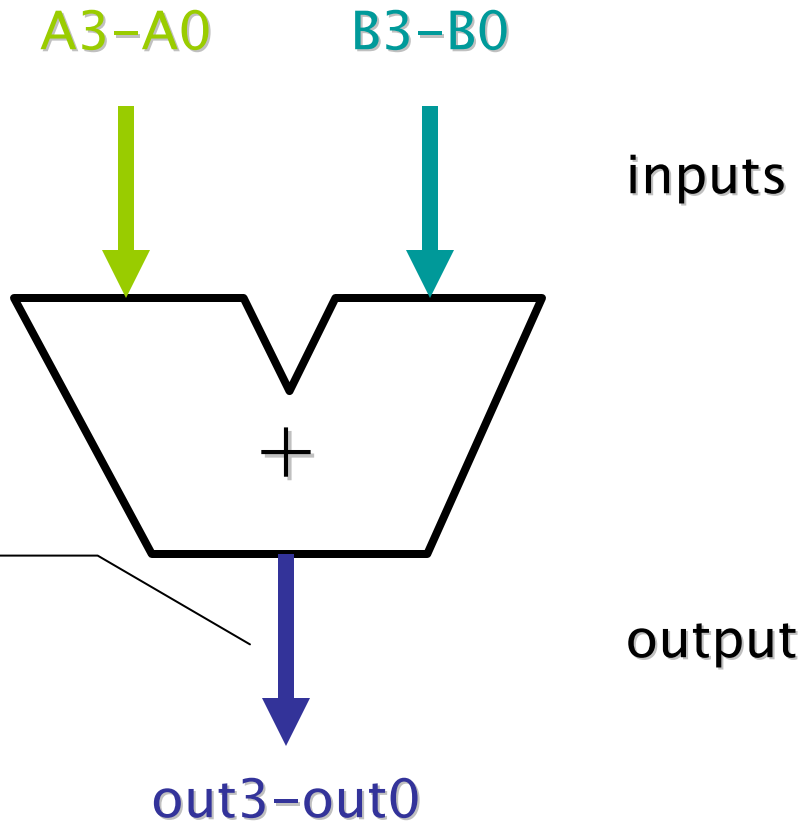


Ripple-carry Adder (II)



Adder

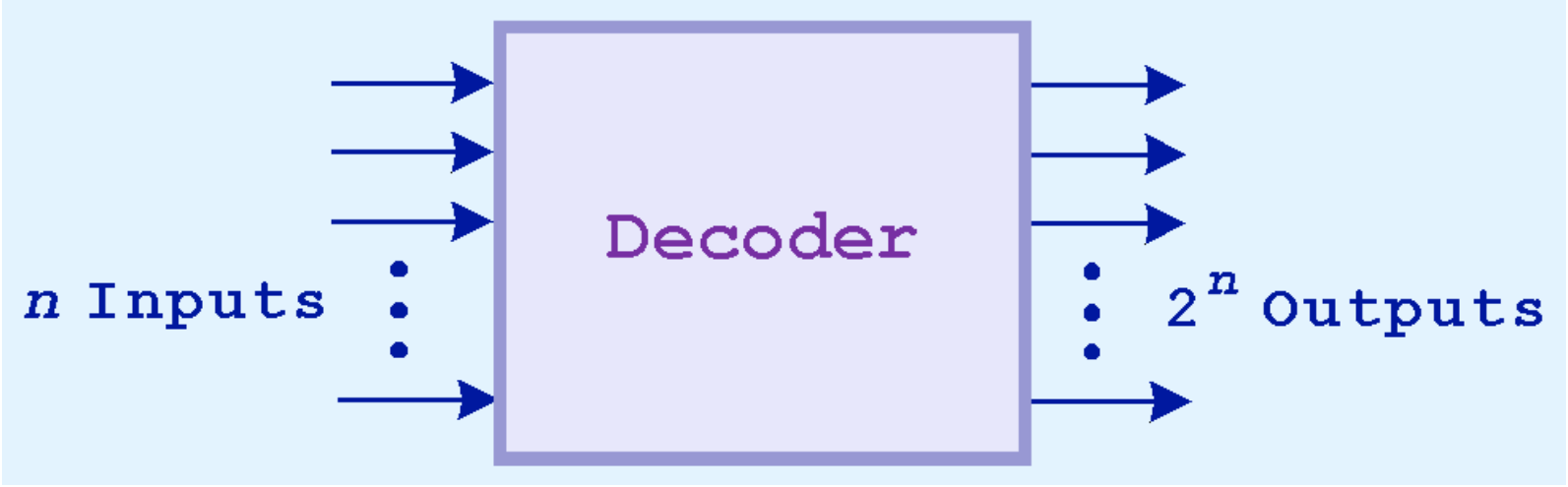
Adders (and other arithmetic circuits) are usually drawn like this in block diagrams



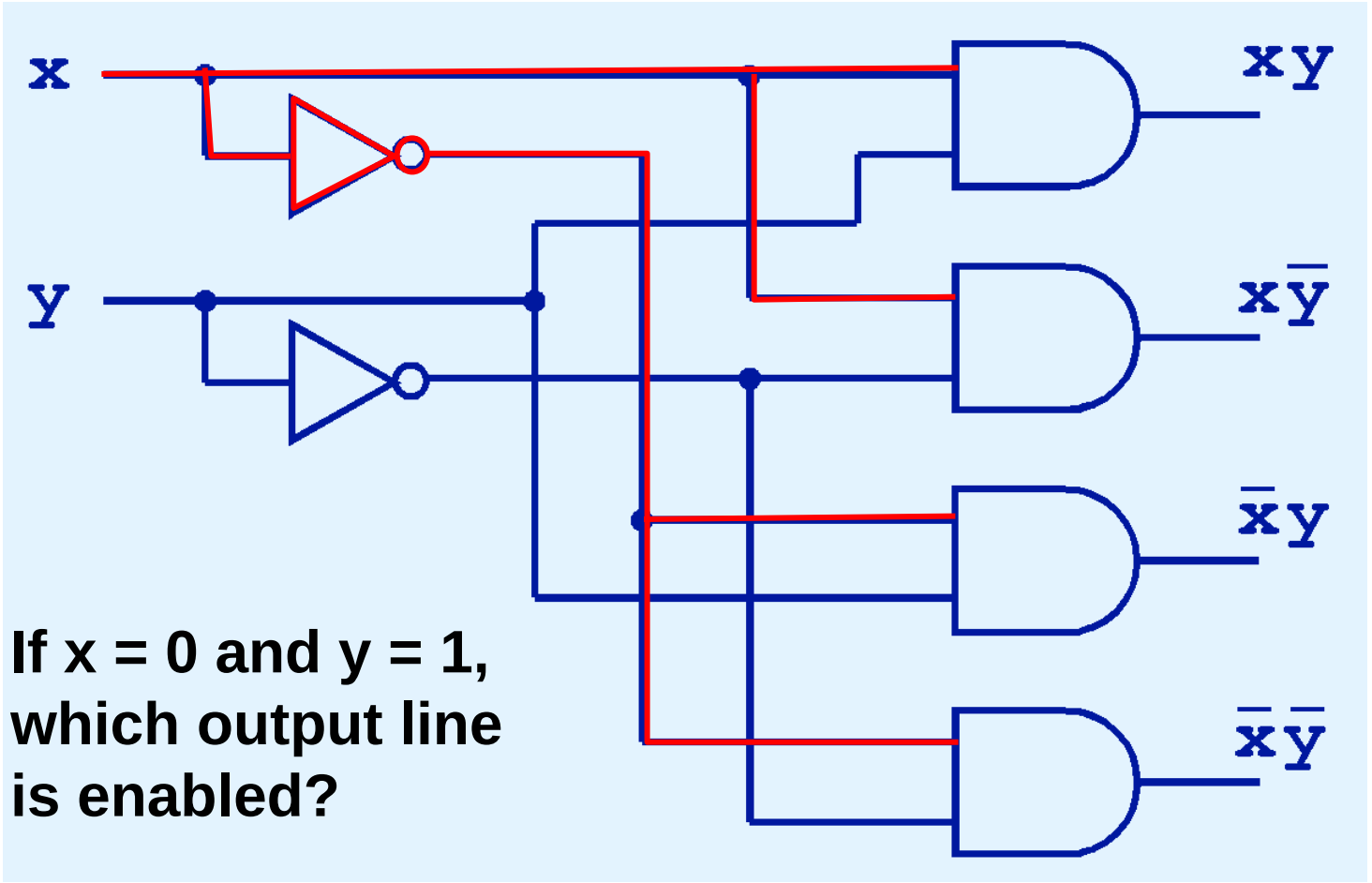
collections of parallel, related wires like this are known as buses; they carry multi-bit values between components

Decoder

- Selects a memory location according a binary value placed on the address lines of a memory bus
- Decoders with n inputs can select any of 2^n locations



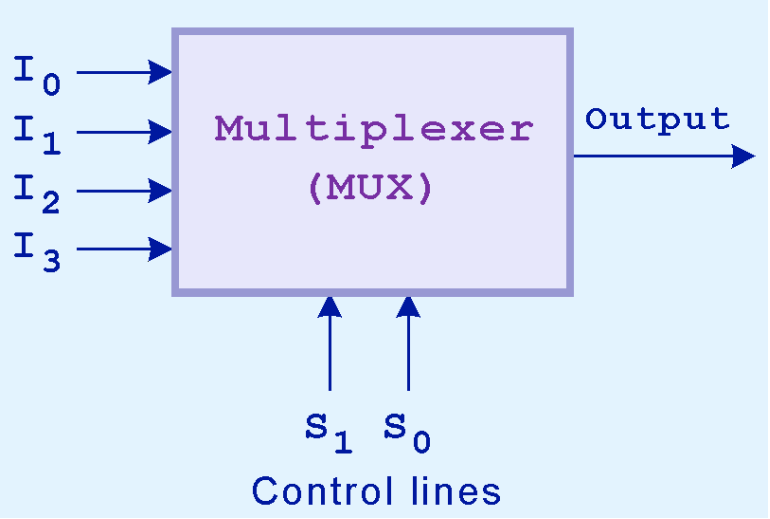
2-to-4 Decoder



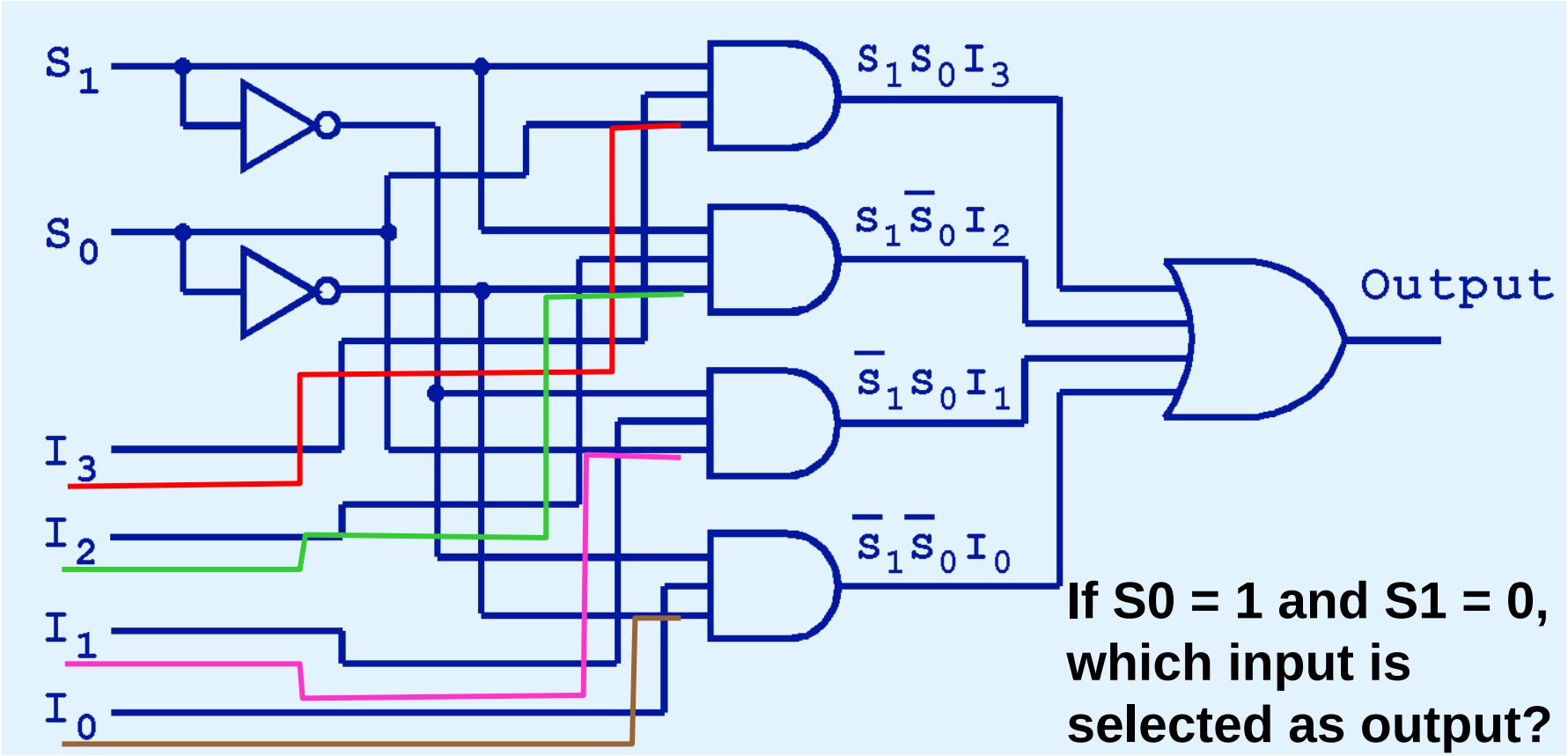
If x = 0 and y = 1, which output line is enabled?

Multiplexer

- A multiplexer does the opposite of a decoder
- Selects a single output from several inputs
  - The particular input chosen for output is determined by the value of the multiplexer's control lines
  - To select among n inputs, $\log_2 n$ control lines are needed



4-to-1 Multiplexer

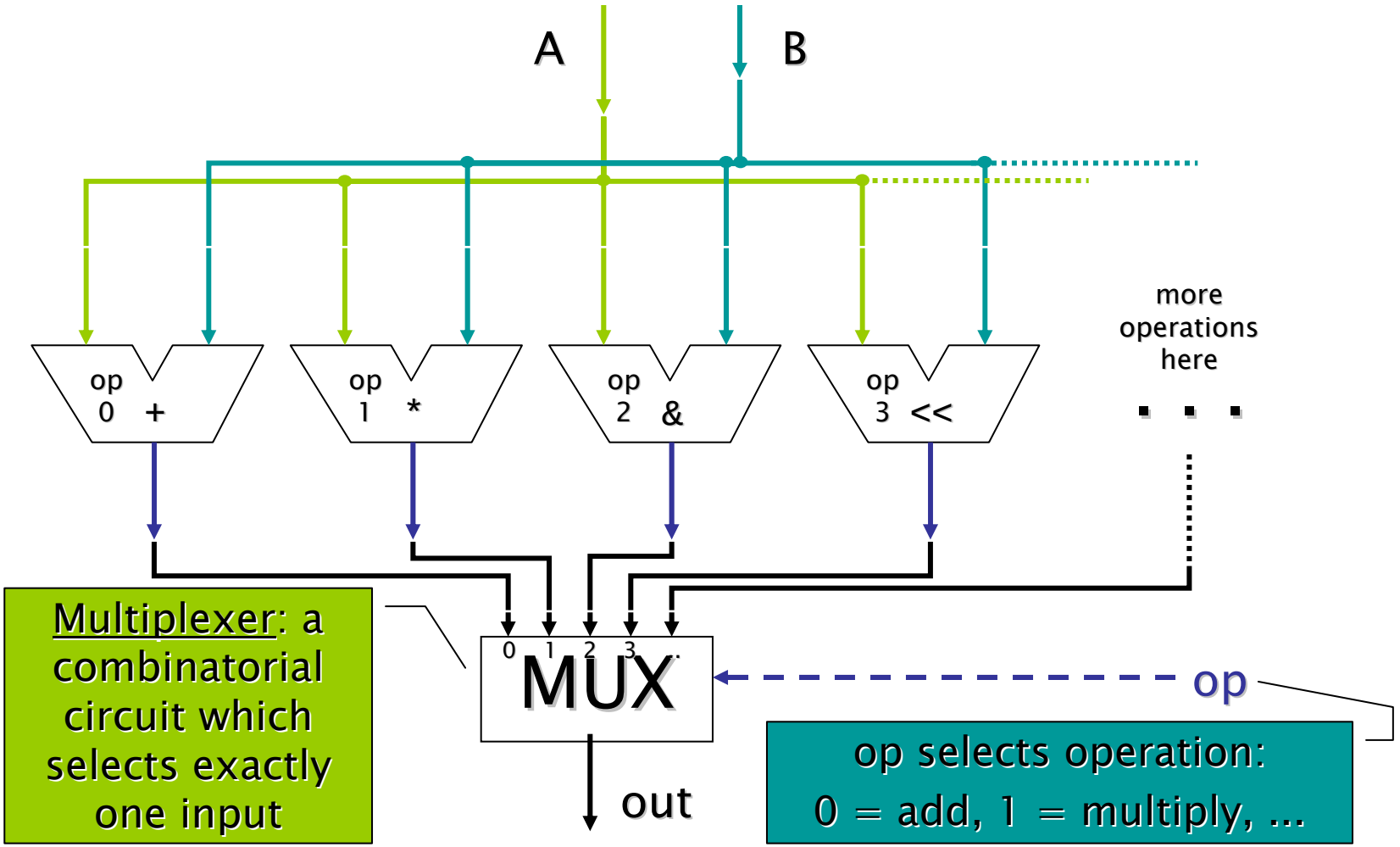


If S0 = 1 and S1 = 0, which input is selected as output?

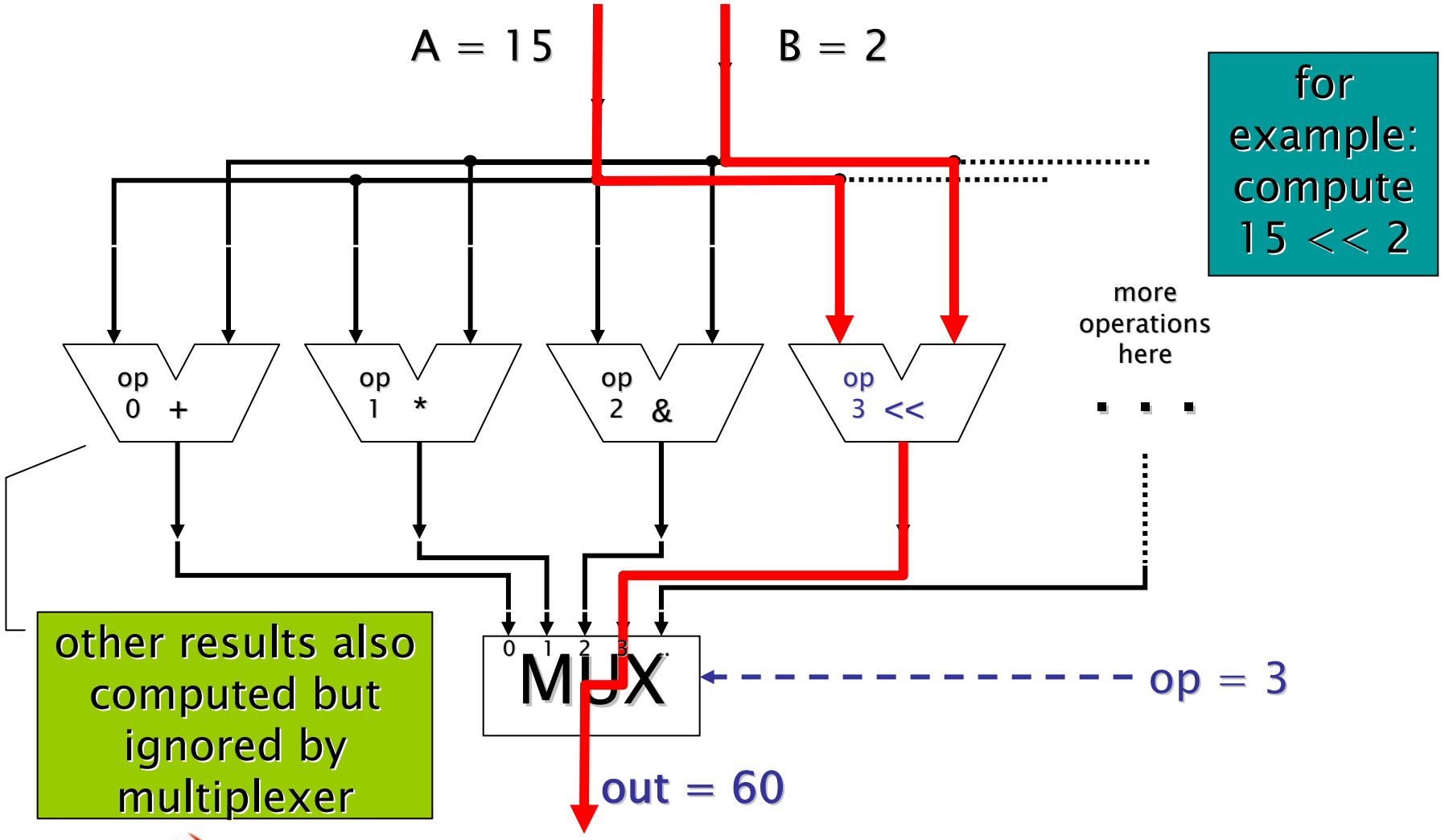
Arithmetic

- Computers need to do more than just addition
  - arithmetic: + – * / %
  - logic: & | ~ << >>
- Need a circuit that can select operation to perform

Arithmetic Logic Unit (ALU)



Arithmetic Logic Unit (ALU)



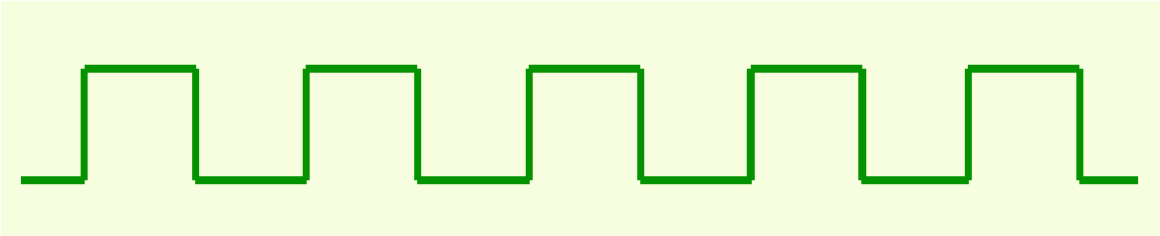
Sequential Logic – Memory

Sequential Logic Circuits

- Combinational logic circuits are perfect for situations which require the immediate application of a Boolean function to a set of inputs
- But, here are times when we need a circuit to change its value with consideration to its current state as well as its inputs
  - These circuits have to “remember” their current state
- *Sequential logic circuits* provide this functionality

Sequencing Events

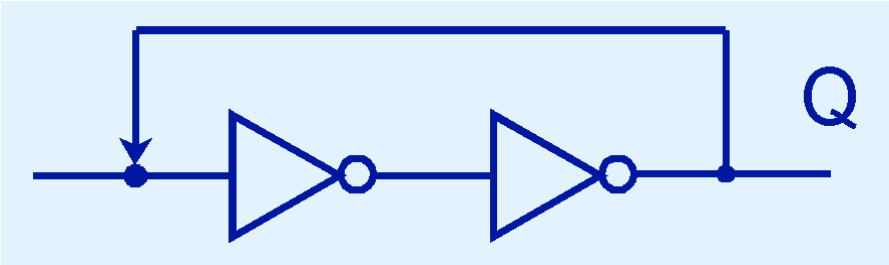
- Sequential logic circuits require a means by which events can be sequenced
  - ➢ State changes are controlled by clocks
    - A “clock” is a special circuit that sends electrical pulses through a circuit
  - ➢ Clocks produce electrical waveforms such as this one



  - ➢ State changes occur in sequential circuits only when the clock ticks

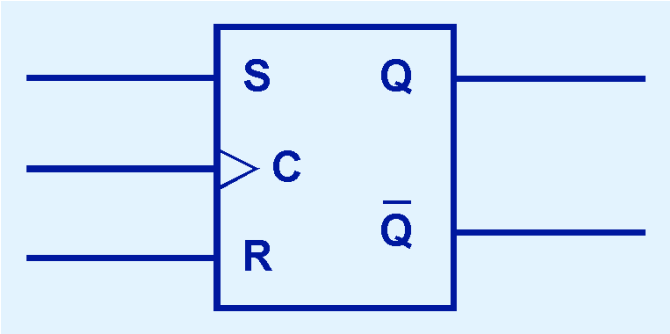
Feedback in Sequential Logic Circuits

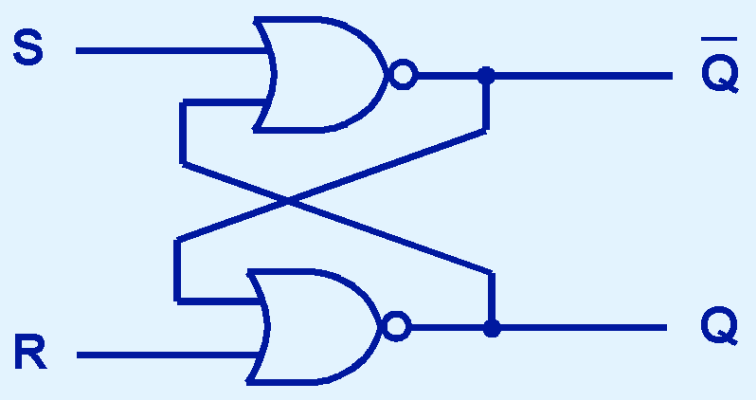
- Sequential circuits rely on feedback to retain their state values
- Feedback in digital circuits occurs when an output is looped back to the input
  - Example,



If Q is 0 it will always be 0, if it is 1, it will always be 1

SR Flip-flop (Set-Reset) (I)



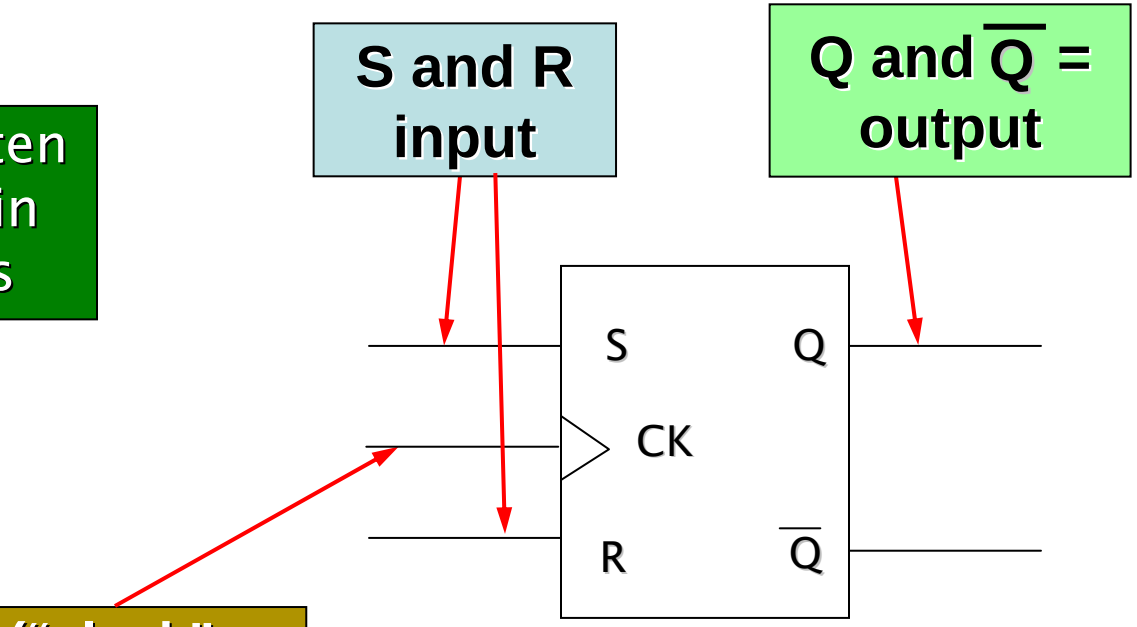


- The behavior of an SR flip-flop is described by a *characteristic table*
  - Q(t) output at time t
  - Q(t+1) output after the next clock pulse

S	R	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	undefined

SR Flip-flop: Block Diagram

Flip-flops are often drawn like this in block diagrams



CK is read/write ("clock" because this input is connected to the computer's processor clock)

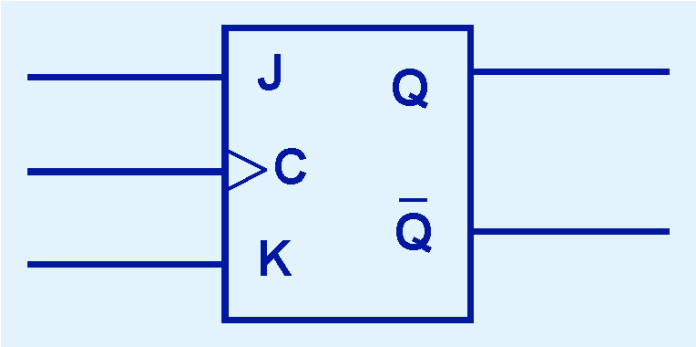
SR Flip-flop (II)

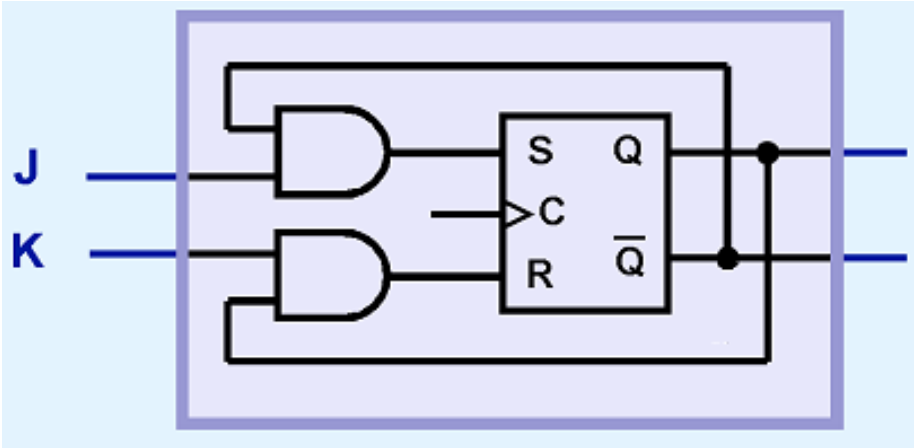
- The SR flip-flop has three inputs: S, R and Q(t)
  - When both S and R are 1, the SR flip-flop is unstable

Present State			Next State
S	R	Q(t)	Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	undefined
1	1	1	undefined

JK Flip-flop (Jack Kilby)

- Modified version of the SR flip-flop to provide a stable state when both inputs are 1

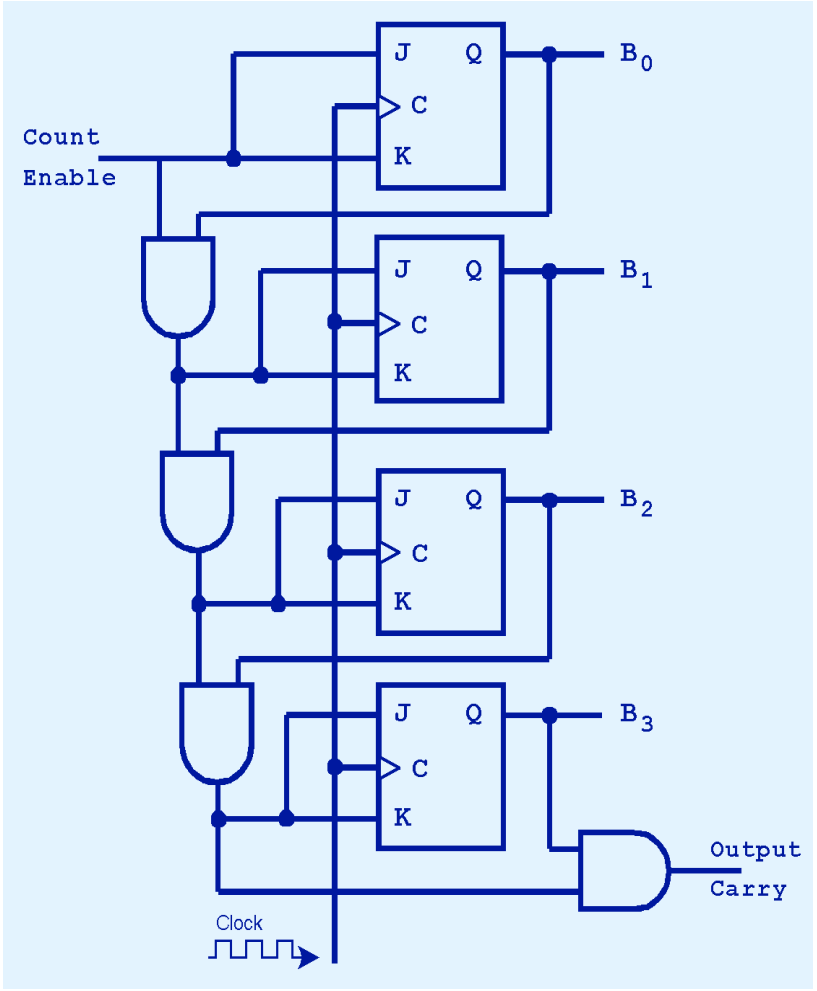




J	K	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\bar{Q}$(t)

JK Flip-flop: Binary Counter

- The low-order bit is complemented at each clock pulse
- Whenever it changes from 1 to 0, the next bit is complemented, and so on through the other flip-flops

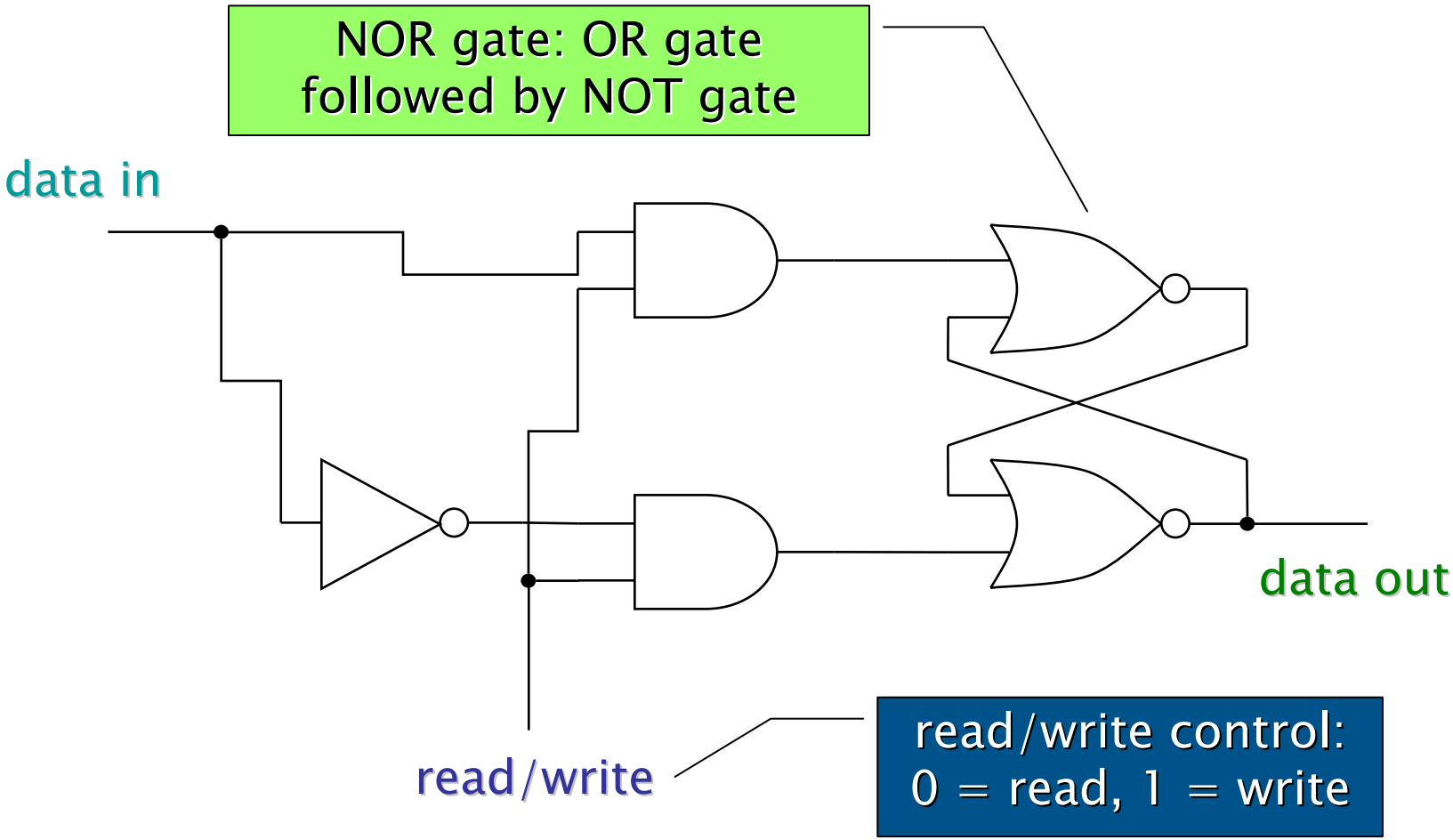


J	K	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\overline{Q}$(t)

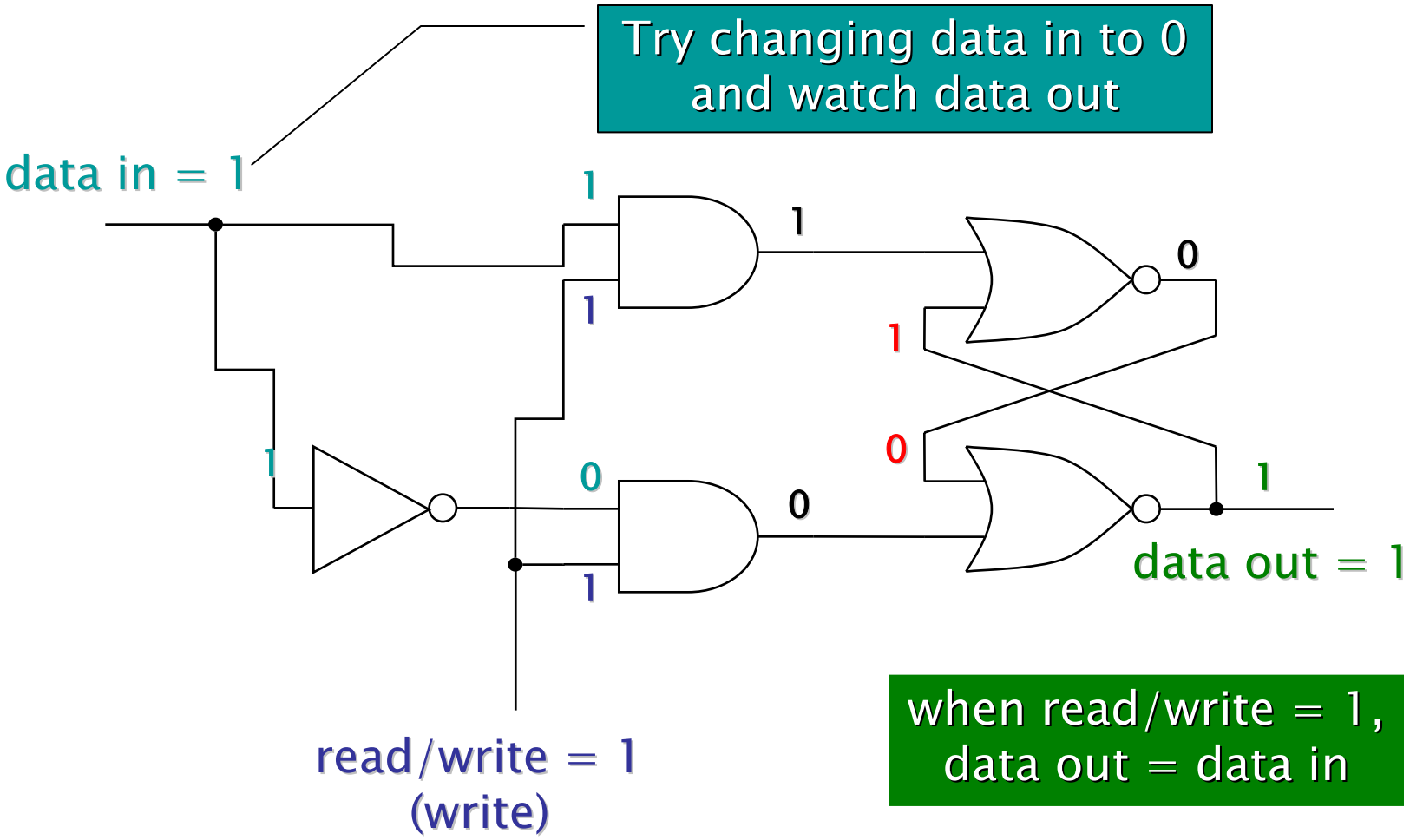
D Flip-flop (Data)

- Fundamental circuit of computer memory
- Used to store 1 bit
- Can be implemented with gates
- Not combinatorial logic
  - because current output may depend on previous state
- Example of sequential logic
  - current output depends on inputs and prior output

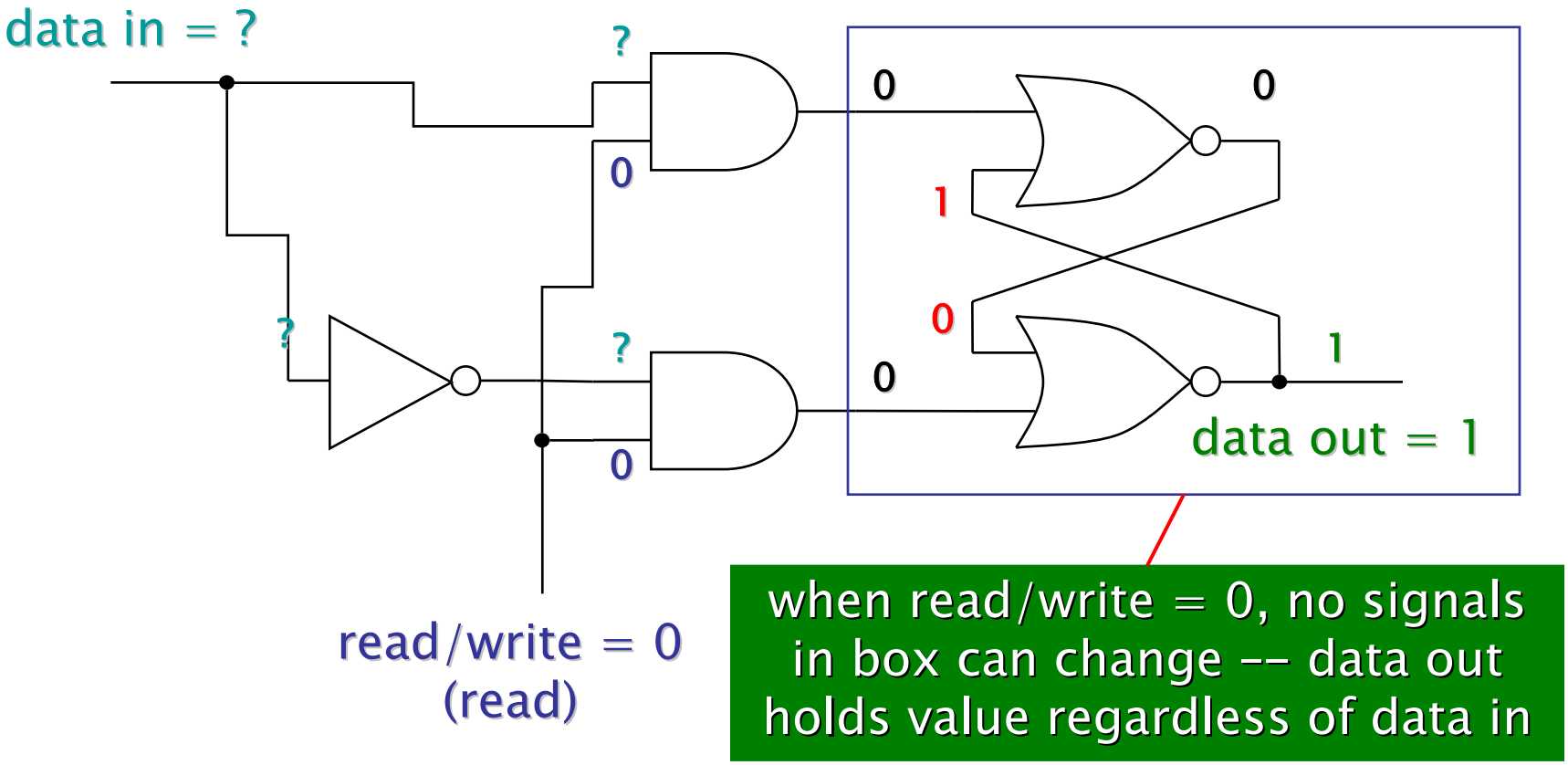
D Flip-flop



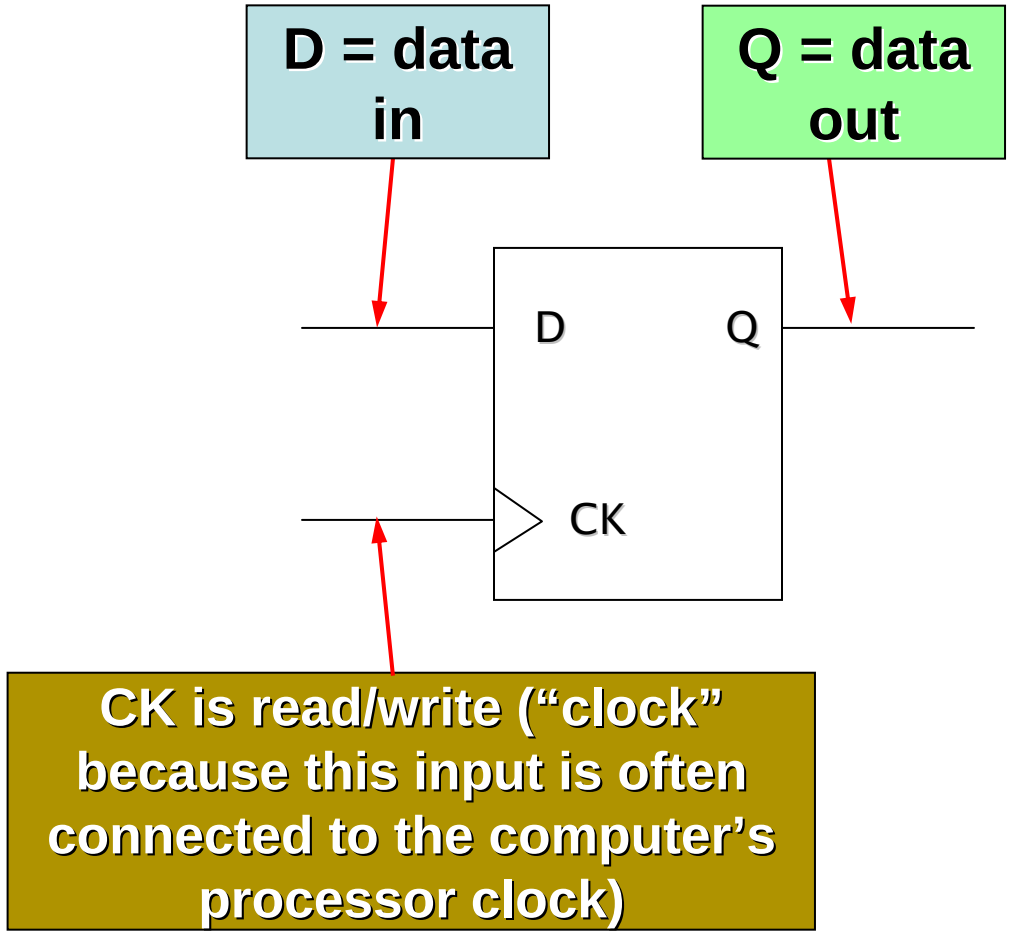
D Flip-flop: Writing



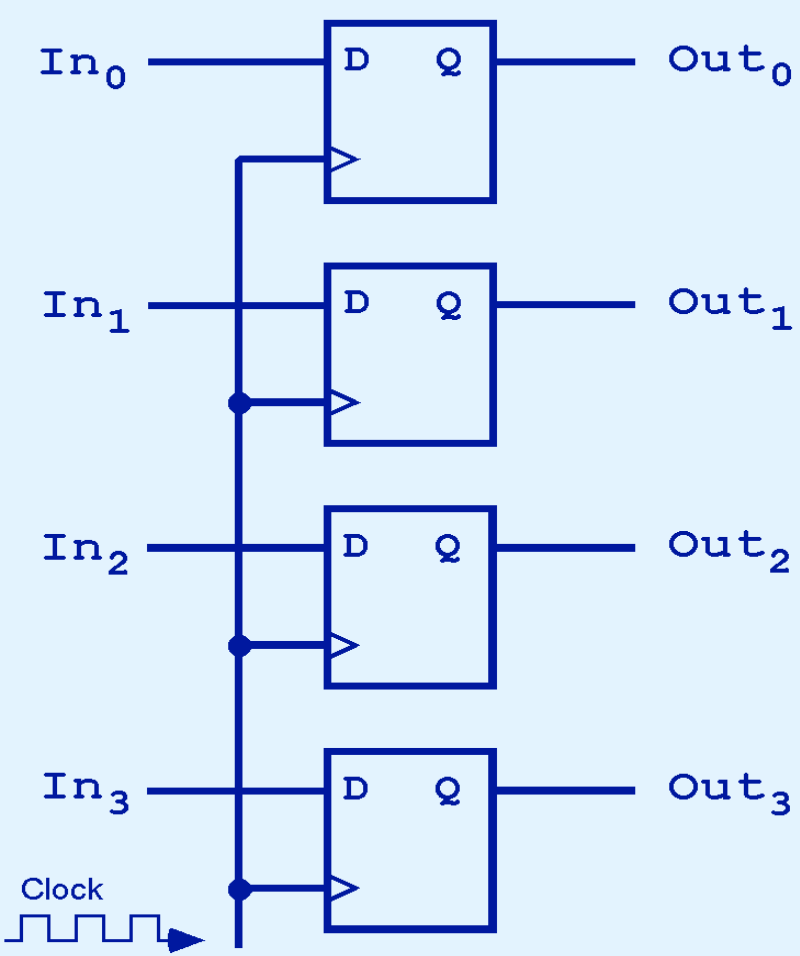
D Flip-flop: Reading



D Flip-flop: Block Diagram



D Flip-flop: 4-bit Register

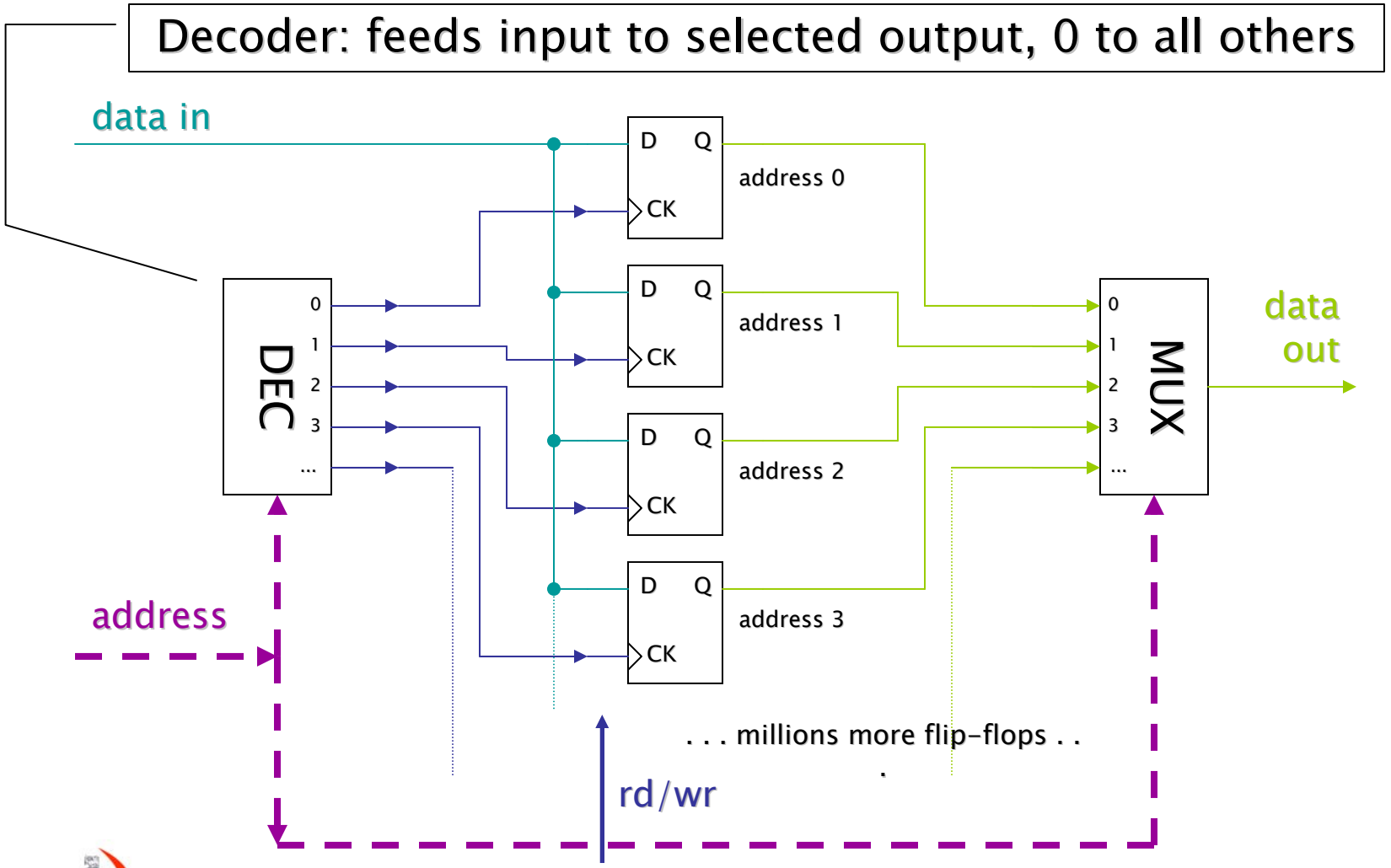


A register stores data inside the CPU

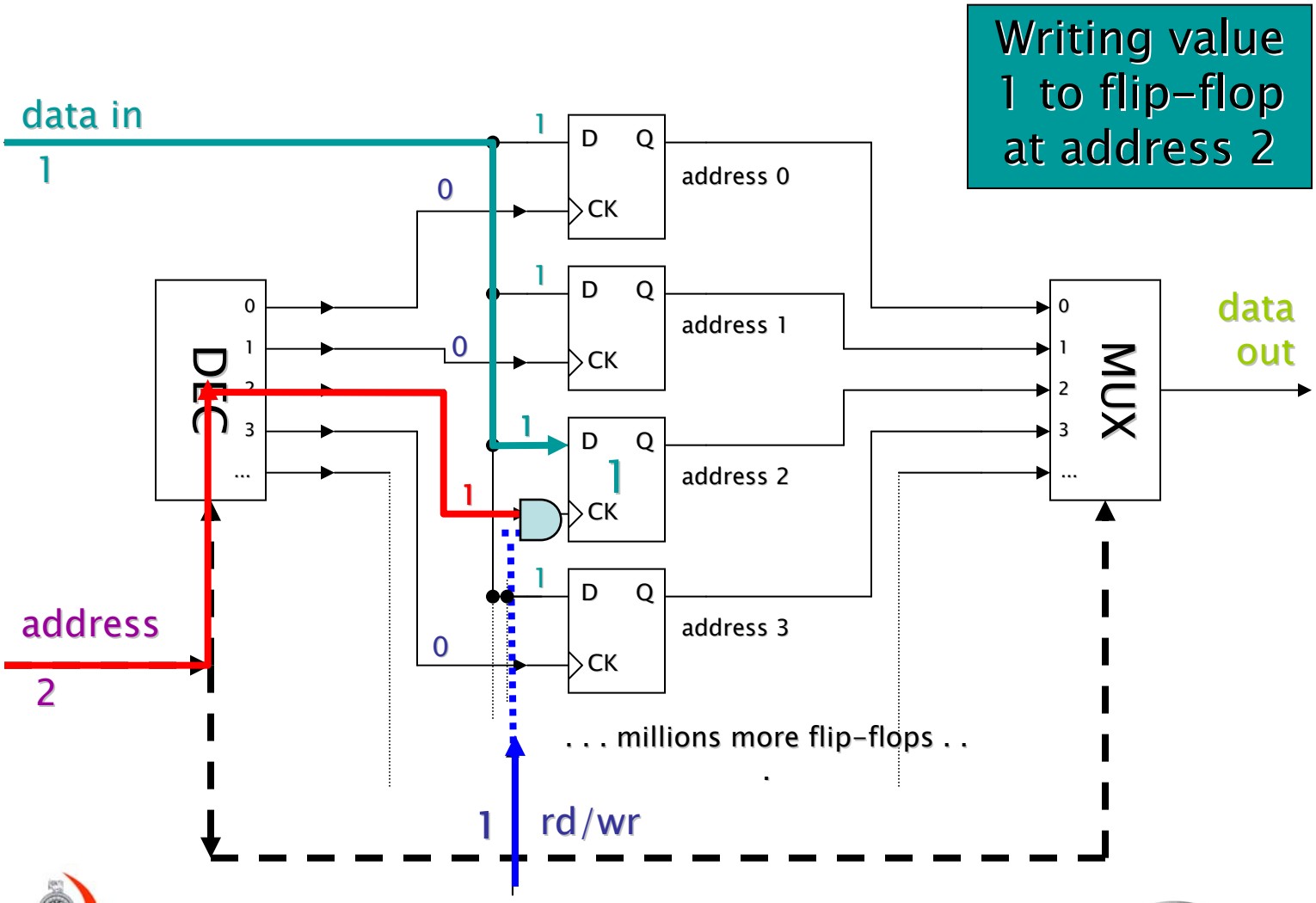
Memory

- Memory can store many bits independently
  - register banks contain many flip-flops
- Need to identify which bit (flip-flop) to read or write
- Give each flip-flop a unique number (address)

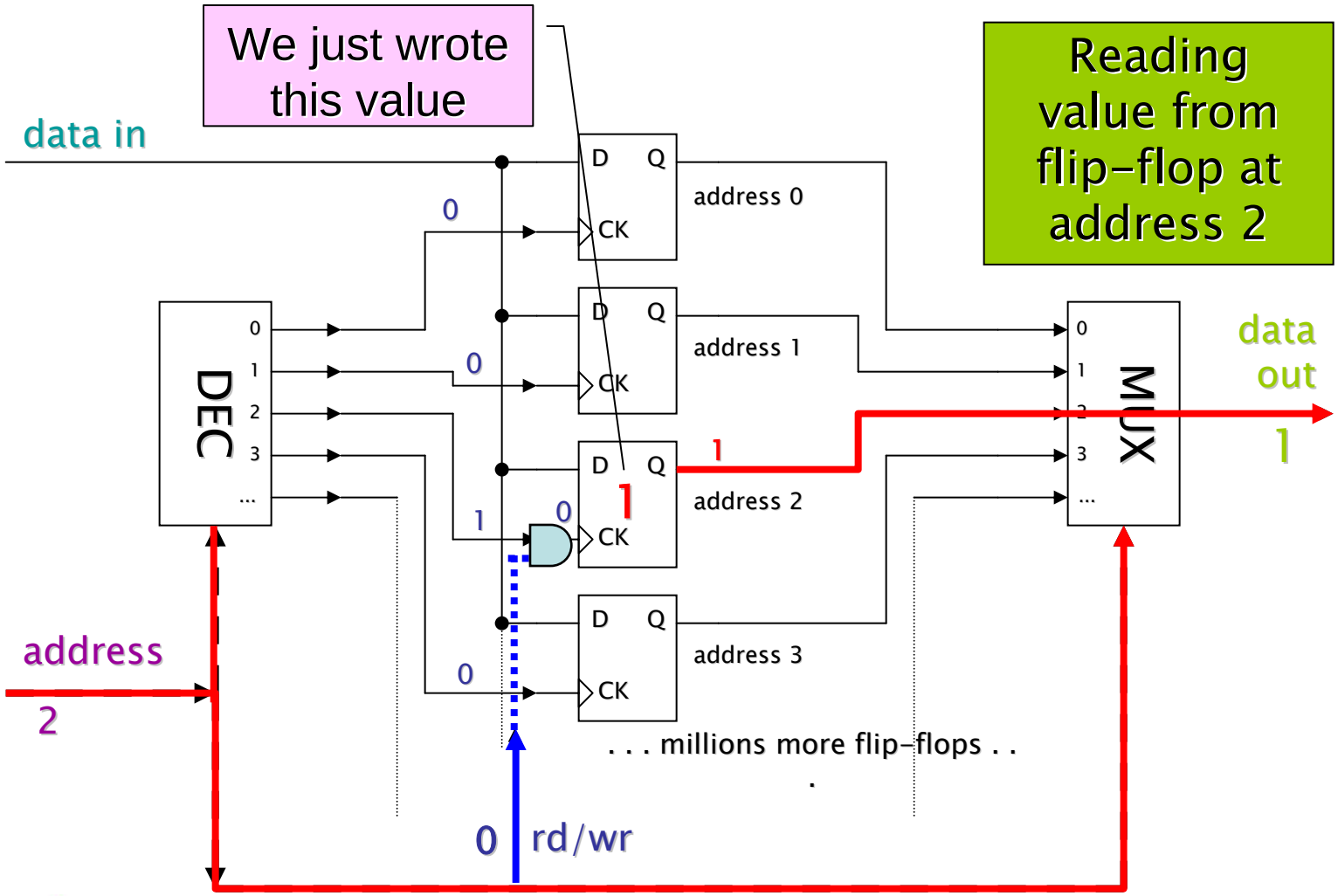
Memory: Circuit Diagram



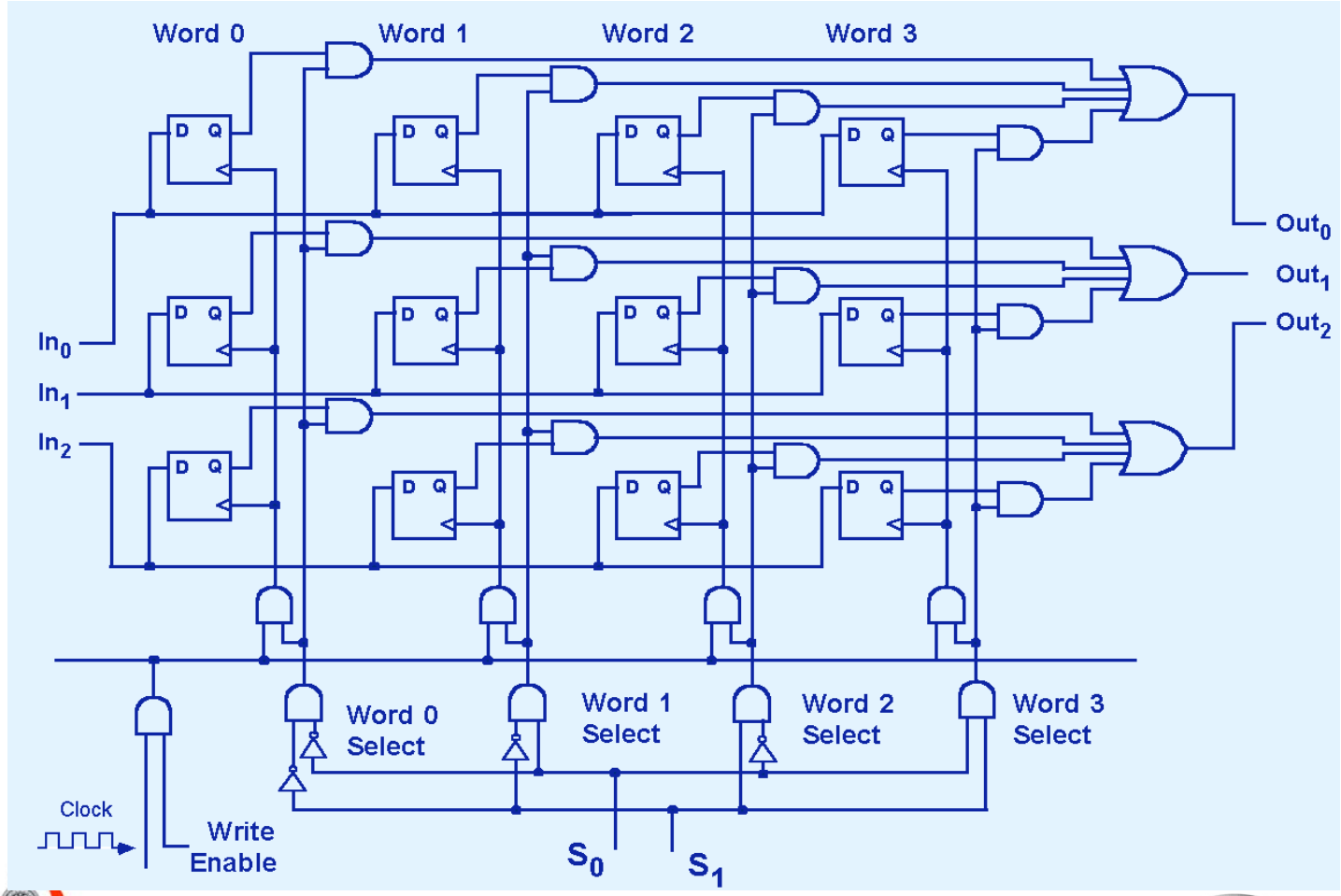
Memory: Writing



Memory: Reading



Memory: 4-words, 3 bits/word



Summary (I)

- Computers are implementations of Boolean logic
- Boolean functions are completely described by Truth Tables
- Logic gates are small circuits that implement Boolean operators
- The basic gates are AND, OR and NOT
- The “universal gates” are NOR and NAND

Summary (II)

Computer circuits consist of combinational logic circuits and sequential logic circuits

- Combinational circuits produce outputs (almost) immediately when their inputs change
- Sequential circuits have internal states as well as combinations of input and output logic
  - The outputs may also depend on the states left behind by previous inputs
  - Sequential circuits require clocks to control their changes of state
  - The basic sequential circuit unit is the flip-flop

Thank You